UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/936,174	12/06/2001	David Naccache	032326-166	9572

21839 7590 03/29/2005

BURNS DOANE SWECKER & MATHIS L L P
POST OFFICE BOX 1404
ALEXANDRIA, VA 22313-1404

EXAMINER	
INGBERG, TODD D	

ART UNIT	PAPER NUMBER
2193	

DATE MAILED: 03/29/2005

IFW

Please find below and/or attached an Office communication concerning this application or proceeding.

SB09

Office Action Summary	Application No.	Applicant(s)	
	09/936,174	NACCACHE ET AL.	
	Examiner	Art Unit	
	Todd Ingberg	2124	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If the period for reply specified above is less than thirty (30) days, a reply within the statutory minimum of thirty (30) days will be considered timely.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *10 September 2001*.

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-26 and 28-31* is/are pending in the application.

4a) Of the above claim(s) *27 and 32* is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1-5,7-13,15,18-20,24-26 and 28-31* is/are rejected.

7)☒ Claim(s) *6,14,16,17,21-23* is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☒ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on *10 September 2001* is/are: a)☐ accepted or b)☒ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☒ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☒ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☒ All b)☐ Some * c)☐ None of:

1.☒ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☒ Information Disclosure Statement(s) (PTO-1449 or PTO/SB/08) Paper No(s)/Mail Date *9/10/2001*.
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____ .
5) ☐ Notice of Informal Patent Application (PTO-152)
6) ☐ Other: _____.

DETAILED ACTION

Claims 1 – 26 and 28-31 have been examined.

In a Preliminary Amendment

Claims 1 – 26 and 28-31 were amended.

Claims 27 and 32 were cancelled.

Priority

1. Receipt is acknowledged of papers submitted under 35 U.S.C. 119(a)-(d), which papers have been placed of record in the file.

Information Disclosure Statement

2. The Information Disclosure Statement (IDS) filed December 6, 2001 has been considered. The reference in French could not be considered.

Oath/Declaration

3. Applicant has elected to use an outdated version of 37 CFR 1.56 "(as amended effective March 16, 1992)". Applicant should use the current form on the USPTO.GOV website when submitting a new Declaration.

Drawings

4. New corrected drawings in compliance with 37 CFR 1.121(d) are required in this application because they are fuzzy and hard to read and will not display properly in a U.S. Patent. Applicant is advised to employ the services of a competent patent draftsperson outside the Office, as the U.S. Patent and Trademark Office no longer prepares new drawings. The corrected drawings are required in reply to the Office action to avoid abandonment of the application. The requirement for corrected drawings will not be held in abeyance.

5. Figure 1 should be designated by a legend such as --Prior Art-- because only that which is old is illustrated. See MPEP § 608.02(g). Corrected drawings in compliance with 37 CFR 1.121(d) are required in reply to the Office action to avoid abandonment of the application. The replacement sheet(s) should be labeled "Replacement Sheet" in the page header (as per 37 CFR 1.84(c)) so as not to obstruct any portion of the drawing figures. If the changes are not accepted by the examiner, the applicant will be notified and informed of any required corrective action in the next Office action. The objection to the drawings will not be held in abeyance.

Specification

6. The abstract of the disclosure is objected to because must be on a separate page. Correction is required. See MPEP § 608.01(b).

7. Preliminary amendment of September 10, 2001 has been entered.

8. The disclosure is objected to because of the following informalities: The spelling of several words is not in the format for United States English, the European spelling of the following must be changed.

European Spelling	United States English Spelling
"analysing"	analyzing
"analysed"	analyzed
"reinitialised"	reinitialized
"reinitialisation"	reinitialization

Correction will benefit the searching of U.S. Patent literature.

Appropriate correction is required.

9. Page 5 of the Specification contains the acronym "FIBs", without the term being fully spelt out. On common meaning is "Secured hash standard, Federal Information Processing Standards Publication (FIBS) 180-1, May 1994". Clarification required with a change to the Specification.

10. The use of the trademark "JAVA" has been noted in this application. It should be capitalized wherever it appears and be accompanied by the generic terminology.

Although the use of trademarks is permissible in patent applications, the proprietary nature of the marks should be respected and every effort made to prevent their use in any manner which might adversely affect their validity as trademarks.

11. The title of the invention is not descriptive. A new title is required that is clearly indicative of the invention to which the claims are directed.

Claim Rejections - 35 USC § 112

12. The following is a quotation of the second paragraph of 35 U.S.C. 112:

> The specification shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention.

13. Claims 8 – 10 are rejected under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention. The problem is the Applicant states the program to be monitored (DATA) . The focus of the claim language should the functionality of the monitor program and how it handles the varies condition presented by the input as it is processed. The Specification clearly supports what the Applicant is attempting to claim. This claim as written is indefinite. Dependent claims are also rejected merely because they are dependent on claim 8.

Claim 8
A method according to Claim 1 wherein, when the program to be monitored provides for at least one jump, the monitoring method is applied separately to sets of instructions in the program which do not include jumps between two successive instructions.

Claim Rejections - 35 USC § 102

14. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

15. Claims 1 –5, 7-13, 15, 18-20, 24-26, 28, 30 and 31 are rejected under 35 U.S.C. 102(b) as being anticipated by USPN # 4,266,272 Berglund et al (**IDS**).

The environment of the invention JAVACARD is not claimed but is vastly different than the environment of the IDS reference

Claim Interpretation

The ***control circuitry*** in the reference IDS is performing the ***monitor*** function of the claimed invention.

Claim 1
IDS anticipates a method for monitoring progress with the execution of a linear sequence of instructions in a computer program (**IDS**, Abstract, control circuitry), comprising the steps of analysing the sequence of instructions transmitted to a processor intended to execute the program being monitored by extracting a data item from each instruction transmitted to the processor (**IDS**, Abstract, check word) and performing a calculation on said data item (**IDS**, Abstract, dynamically calculated), and verifying, the result of this analysis by comparing the result of said calculation to reference data (**IDS**, Abstract, local storage register vs. ALU), recorded with said program, wherein the reference data comprises a value pre-established so as to correspond to the result of the analysis produced during the monitoring method only if all the instructions in the sequence of instructions have actually been analysed during the running of the program (**IDS**, Abstract, control storage).
Claim Interpretation
The limitation "of a linear sequence of instructions" is not given patentable weight because it is dependent on the form of the input. Not part of the invention. It is treated as data.

Claim 2
A method according to Claim 1, wherein the verification of the result of the analysis is caused by an instruction placed at a predetermined location in the program to be monitored (as per claim 1 a register is a predetermined location), said instruction containing the reference data relating to a set of instructions whose correct execution is to be monitored (registers are inherently related to the instruction being processed).

Claim 3
A method according to Claim 1 wherein, when the instructions of the set of instructions to be monitored are in the form of a value, said analysis of the instructions is carried out by using these instructions as a numerical value. (Interpretation – all values are in binary format – this is inherent).

Claim 4
A method according to Claim 1, comprising the steps of:
- during the preparation of the program to be monitored (as per claim 1):
- incorporating, in at least one predetermined location in a sequence of instructions (as per claim 1) in the program, a reference value established according to a predetermined rule applied to identifiable data in each instruction to be monitored (as per claim 1, identification of words), and during the execution of the program to be monitored (as per claim 1):
- obtaining said identifiable data in each instruction received for execution (IDS, fetch col 9, 10-30),
- applying said predetermined rule to said identifiable data thus obtained in order to establish a verification value (as per claim 1), and
- verifying that this verification value actually corresponds to the reference value recorded with the program (as per claim 1).

Claim 5
A method according to Claim 1, further comprising a step of interrupting the flow of the program if the analysis reveals that the program being monitored has not been run as expected. (**IDS**, Figure #4, Result ERROR from result branch).

Claim 7
A method according to Claim 1 wherein the set of instructions to be monitored does not include jumps in its expected flow.
Claim Interpretation
The limitation "set of instructions to be monitored does not include jumps in its expected flow" is not given patentable weight because it is dependent on the form of the input. Not part of the invention. It is treated as data.

Claim 8
A method according to Claim 1 wherein, when the program to be monitored provides for at least one jump, the monitoring method is applied separately to sets of instructions in the program which do not include jumps between two successive instructions (**IDS,** col 9, lines 10 – 40).

Claim Interpretation
The limitation "program to be monitored provides for at least one jump" is not given patentable weight because it is dependent on the form of the input. Not part of the invention. It is treated as data.

Claim 9
A method according to Claim 8, wherein, when the program to be monitored includes an instruction for a jump dependent on the manipulated data, the monitoring method is implemented separately for a set of instructions which precedes the jump, and for at least one set of instructions which follows said jump. As per claim 8.

Claim 10
A method according to Claim 9, wherein, for a set of instructions providing for a jump, an instruction which controls this jump is integrated in said set of instructions for the purpose of obtaining a verification value for thus set of instructions before executing the jump instruction. as per claim 8.

Claim 11
A method according to Claim 1 wherein the analysis is reinitialised before each new monitoring of a sequence of instructions to be monitored. (**IDS,** cycle and incrementer, col 3, lines 40 – 60)

Claim 12
A method according to Claim 11, wherein the reinitialisation of the analysis of each new monitoring includes the step of erasing or replacing a verification value obtained during a previous analysis. As per claim 11 depending on cycle determination.

Claim 13
A method according to Claim 11 wherein the reinitialisation of the monitoring analysis is controlled by the software itself. (Interpretation – the control circuitry and software being executed has a functional relationship – This is deemed inherent and related to Examiner's note above)

Claim 15
A method according to Claim 1 wherein the analysis includes the step of calculating, for each instruction under consideration following a previous instruction, the result of an operation on both a value obtained of the instruction in question and the result obtained by the same operation performed on the previous instruction. As per claim 1.

Claim 18
A method according to Claim 1 wherein the analysis includes the step of obtaining a comparison value by calculating successive intermediate values as the data of the respective instructions are obtained. (IDS, Abstract, last sentence words is plural).

Claim 19

A method according to Claim 1 wherein the analysis comprises a step of saving each data item necessary for verification, obtained from instructions in the set of instructions to be monitored as they are executed, and performing a calculation of a verification value from these data only at the necessary time, once all the necessary data have been obtained. (as per claim 1 and details of fetch col 3, lines 10-30).

Claim 20

IDS anticipates a device for monitoring progress with the execution of a series of instructions of a computer program, comprising means for analysing the sequence of instructions transmitted to the processor intended to execute the program being monitored by extracting a data item from each instruction transmitted to the processor and performing a calculation on said data item, and means for verifying the result of this analysis by comparing the result of said calculation to reference data recorded with said program, wherein the reference data comprises a value pre-established so as to correspond to the result of the analysis produced during monitoring only if all the instructions in the sequence of instructions have actually been analysed during the running of the program. As per claim 1.

Claim Interpretation

The limitation "a series of instructions of a computer program" is not given patentable weight because it is dependent on the form of the input. Not part of the invention. It is treated as data.

Claim 24

A device according to Claim 20 that is integrated into a programmed device containing said program to be monitored. (**IDS**, Abstract, Control Circuitry).

Claim 25

A device according to Claim 20 that is integrated into a program execution device. (**IDS**, Abstract, Control Circuitry).

Claim 26

IDS anticipates a program execution device that executes a series of instructions of a computer program, comprising means for analysing the sequence of instructions transmitted for execution by extracting a data item from each instruction and performing a calculation on said data item, and means for verifying the result of this analysis by comparing the result of said calculation to reference data recorded with the program to be monitored, wherein the reference data comprises a value pre-established so as to correspond to the result of the analysis produced during monitoring only if all the instructions in the sequence of instructions have actually been analysed during the running of the program. As per claim 1.

Claim Interpretation

A. The limitation "a series of instructions of a computer program" is not given patentable weight because it is dependent on the form of the input. Not part of the invention. It is treated as data.
B. In a similar fashion. The limitation "correspond to the result of the analysis produced during monitoring only if all the instructions in the sequence of instructions have actually been analysed during the running of the program" can be dependent on the input. If the program is only s few statement which all statements are to execute the claim limitations are input dependent. The

claim limitations outside the fact a monitor function is present have not performed a non required step to distinguish it from a monitor.

Claim 28
IDS anticipates a programmed device containing a series of recorded instructions and a fixed memory containing reference data pre-established as a function of data contained in said instructions for analysis and verification of the sequence of instructions, wherein the reference data comprises a value pre-established so as to correspond to the result of the analysis produced during monitoring only if all the instructions in the sequence of instructions have actually been analyzed during the running of the program. as per claim 1.
Claim Interpretation
A. The limitation "a series of recorded instructions" is not given patentable weight because it is dependent on the form of the input. Not part of the invention. It is treated as data.

Claim 30
A device according to Claim 28 wherein the reference data are recorded in the form of a prewired value or values fixed in memory. (**IDS** , Abstract, last sentence).
Claim Interpretation
The presence of the OR in the limitations. the Examiner elects to reject the underlined limitation above.

Claim 31
A device for programming a programmed device according to Claim 28, comprising means for entering, in at least one predetermined location in a sequence of instructions in the program, a reference value calculated according to a preestablished mode from data included in each instruction in a set of instructions whose execution is to be monitored. As per claim 1.

Claim Rejections - 35 USC § 103

16. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

17. Claim 29 is rejected under 35 U.S.C. 103(a) as being unpatentable over IDS in view of USPN # 6,402,028.

Claim 29

IDS teaches a device but IDS does not teach the device is a smart card. according to Claim 28, wherein said device is a smart card. USPN 6,402,028 teaches the production of Smart Cards where the logic is on the card. therefore, it would have been obvious to one of ordinary skill in the art at the time of invention, to combine IDS with 6,402,028 because logic control for Smart cards makes Smart Cards more reliable.

Allowable Subject Matter

18. Claims 6, 14, 16, 17 and 21 – 23 are objected to as being dependent upon a rejected base claim, but would be allowable if rewritten in independent form including all of the limitations of the base claim and any intervening claims. The bold and underlined limitations below indicate limitations not found in the prior art of record.

Claim 6
A method according to Claim 1, further comprising an invalidation step for future use of the device comprising the monitored program if **said analysis reveals a predetermined number of times that the program being monitored has not run in the expected manner**.

Claim 14
A method according to Claim 1 wherein the analysis produces a verification value obtained as the last value **in a series of values which is made to change successively with the analysis of each of the analysed instructions of the set of instructions, thus making it possible to contain an internal state of the running of the monitoring method and to follow its changes**.

Claim 16
A method according to Claim 1 wherein the analysis includes the step **of recursively applying a hash function to values obtained of each monitored instruction**, starting from a last initialisation performed.

Claim 17
A method according to Claim 1 wherein the analysis includes the step of making a verification value change by **performing a redundancy calculation on all the operating codes and the addresses executed since the last initialisation was carried out**.

Claim 21
A device according to Claim 20, further including a register for recording intermediate results in **a calculation in a chain carried out by the analysis means in order to obtain a verification value**.

Claim 22
A device according to Claim 21, further comprising means for recording a predetermined value or resetting the register under the control of an instruction transmitted during the execution of a program to be monitored. (Dependent on claim 21)

Claim 23
A device according to Claim 20, further comprising means for counting the number of unexpected events in the program being monitored, as determined by the analysis means, and **means for invalidating the future use of the program to be monitored if this number reaches a predetermined threshold.**

Conclusion

19. The prior art made of record and not relied upon is considered pertinent to applicant's disclosure.

US Patent Literature

A. 6,402,028 – Deals with mass production of Smart Cards Column 4 covers JAVACARD technology.

B. 6,668,325 – Employs an obfuscation technique on a section of code. Environment is distributed.

C. 5,974,549 – Monitor is implemented via Dynamic Link Library (DLL).

D. 6,546,546 – Appears to be dependent on the extensible operating system disclosed (PARAMECIUM).

E. 6,092,120 – Focus on class loaders.

F. 6,327,700 – Based on Profile data.

G. 6,557,168 – Monitor is included at class level not at low level as per disclosed invention.

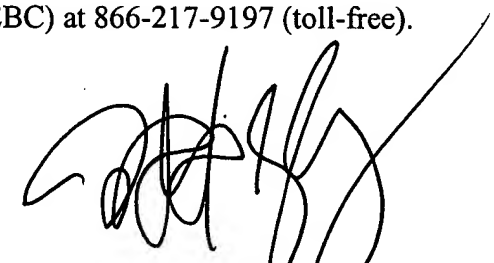
H. 6,275,938 – The monitor environment runs at operating system level not processor level as disclosed invention.

Correspondence

20. Any inquiry concerning this communication or earlier communications from the examiner should be directed to Todd Ingberg whose telephone number is (571) 272-3723. The examiner can normally be reached on during the work week..

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Kakali Chaki can be reached on (571) 272-3719. The fax phone number for the organization where this application or proceeding is assigned is 703-872-9306.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).



Todd Ingberg
Primary Examiner
Art Unit 2124

TI

09/936174

SHEET 1 OF 1

Substitute for form 1449A/PTO	ATTORNEY'S DKT NO.	APPLICATION NO.
INFORMATION DISCLOSURE STATEMENT BY APPLICANT	032326-166	~~Unassigned~~ 09/936,174
	APPLICANT: GIRARD Pierre et al.	
	FILING DATE: September 6, 2001	GROUP: ~~Unassigned~~ 2129

U.S. PATENT DOCUMENTS

Examiner Initials	U.S. Patent Document Number	Kind Code (if known)	Name of Patentee or Applicant of Cited Document	Date of Publication (MM-DD-YYYY)
PF	4, 266, 272A		Berglund et al.	05/05/1981

FOREIGN PATENT DOCUMENTS

Examiner Initials	Foreign Patent Document Number	Kind Code (if known)	Country	Date of Publication (MM-DD-YYYY)	Translation Yes	Translation no
French	~~0012794A~~		~~Europe~~	~~07/09/1980~~		X

NON PATENT LITERATURE DOCUMENTS

Examiner Initials	Include name of author (in CAPITAL LETTERS), title of the article (when appropriate), title of the item (book, magazine, journal, serial, symposium, catalog, etc.), date, page(s), volume-issue number(s), publisher, city and/or country where published.

Examiner Signature		Date Considered	1/28/04

EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609; draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant. SEND TO: Assistant Commissioner for Patents, Washington, D.C. 20231.

(05/01)

Notice of References Cited	Application/Control No. 09/936,174	Applicant(s)/Patent Under Reexamination NACCACHE ET AL.	
	Examiner Todd Ingberg	Art Unit 2124	Page 1 of 3

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-6,862,684	03-2005	DiGiorgio, Rinaldo	713/163
	B	US-6,402,028	06-2002	Graham et al.	235/380
	C	US-6,418,420	07-2002	DiGiorgio et al.	705/40
	D	US-6,615,264	09-2003	Stoltz et al.	709/227
	E	US-6,581,206	06-2003	Chen, Zhiqun	717/143
	F	US-6,023,764	02-2000	Curtis, Bryce Allen	713/200
	G	US-5,983,348	11-1999	Ji, Shuang	713/200
	H	US-6,092,120	07-2000	Swaminathan et al.	709/247
	I	US-5,974,549	10-1999	Golan, Gilad	713/200
	J	US-6,802,054	10-2004	Faraj, Mazen	717/128
	K	US-6,546,546	04-2003	Van Doorn, Leendert Peter	717/114
	L	US-6,557,168	04-2003	Czajkowski, Grzegorz J.	717/151
	M	US-6,199,181	03-2001	Rechef et al.	714/38

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	JAVA 2 Complete, SYBEX, Steven Holzner, pages 3-28, 1999
	V	"A New Public Key Crytosystem Based on Higher Residues", David Naccache et al, ACM 1998, pages 59 - 66
	W	"Twin Signatures:An Alternative to the Hash-and-Sign Paradigm", David Naccache et al, ACM 2001, pages 20 - 27
	X	"Batch Exponentiation A Fast DLP-based Signature Generation Strategy", David M'Raithi and David Naccache, ACM, 1996, pages 58 - 61

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

Notice of References Cited	Application/Control No. 09/936,174	Applicant(s)/Patent Under Reexamination NACCACHE ET AL.	
	Examiner Todd Ingberg	Art Unit 2124	Page 2 of 3

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-6,275,938	08-2001	Bond et al.	713/200
	B	US-6,327,700	12-2001	Chen et al.	717/127
	C	US-6,668,325	12-2003	Collberg et al.	713/194
	D	US-6,510,352	01-2003	Badavas et al.	700/19
	E	US-5,991,414	11-1999	Garay et al.	713/165
	F	US-5,933,498	08-1999	Schneck et al.	705/54
	G	US-6,314,409	11-2001	Schneck et al.	705/54
	H	US-6,859,533	02-2005	Wang et al.	380/28
	I	US-5,347,581	09-1994	Naccache et al.	380/30
	J	US-5,452,357	09-1995	Naccache, David	713/172
	K	US-6,698,662	03-2004	Feyt et al.	235/492
	L	US-2003/0079127	04-2003	Bidan et al.	713/172
	M	US-2004/0088555	05-2004	Girard et al.	713/192

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	"PicoDBMS:Scaling Down Database Technique For The Smartcard", Philippe Pucheral et al, VLDL Journal, 2001, pages 120-132
	V	"Implementation for Coalesced Hashing", Jeffrey Scott Vitter Brown University, ACM, December 1982, pages 911-926
	W	"Optimal Arrangement of Keys in Hash Table", Ronald L. Rivest, ACM, April 1978, pages 200-209
	X	"Code Optimization Techniques for Embedded DSP Microprocessors", Stan Liao et al, ACM, 1995, 6 pages

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

Notice of References Cited	Application/Control No. 09/936,174	Applicant(s)/Patent Under Reexamination NACCACHE ET AL.	
	Examiner Todd Ingberg	Art Unit 2124	Page 3 of 3

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-2002/0174309	11-2002	Naccache et al.	711/163
	B	US-2003/0188170	10-2003	Bidan et al.	713/182
	C	US-6,279,123	08-2001	Mulrooney, Timothy J.	714/35
	D	US-6,507,904	01-2003	Ellison et al.	712/229
	E	US-6,065,108	05-2000	Tremblay et al.	712/201
	F	US-6,021,469	02-2000	Tremblay et al.	711/125
	G	US-6,014,723	01-2000	Tremblay et al.	711/1
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	"Fundemental Technique for Order Optimization", David Simmen et al, ACM, 1996, pages 57 - 67
	V	
	W	
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

World's #1 Java Book Value

1,000 Pages
ONLY $19.99! U.S.

Java™ 2 COMPLETE

- Learn Java Essentials
- Build Interactive Web Applications with the JDK and Work with Java Objects
- Create Sophisticated GUIs with JFC Swing Components and the 2D API
- Learn About Java Beans

SYBEX

Java™ 2 Complete

SYBEX SAN FRANCISCO ▸ PARIS ▸ DÜSSELDORF ▸ SOEST ▸ LONDON

Associate Publisher: Gary Masters

Contracts and Licensing Manager: Kristine O'Callaghan

Acquisitions & Developmental Editors: Denise Santoro and Maureen Adams

Project Editor: Gemma O'Sullivan

Compilation Editor: Suzanne Goraj

Editors: Lisa Duran, Kim Wimpsett, Maureen Adams, Shelby Zimmerman, Alison Moncrieff, Steve Gilmartin, Laura Arendal, Krista Reid-McLaughlin

Compilation Technical Editor: Kirky Ringer

Technical Editors: Matthew Fielder, Kirky Ringer, John Zukowski

Book Designer: Maureen Forys, Happenstance Type-O-Rama

Graphic Illustrators: Tony Jonick, Patrick Dintino, Inbar Berman

Electronic Publishing Specialists: Cyndy Johnsen and Maureen Forys

Production Coordinator: Susan Berge

Indexer: Nancy Guenther

Cover Designer: Design Site

Cover Illustrator: Jack D. Myers

Library of Congress Card Number: 99-60006
ISBN: 0-7821-2468-2

Manufactured in the United States of America

10 9 8 7 6 5 4

TRADEMARKS:

SYBEX has atter
descriptive term

The author and
is based upon fi
based upon pre
publisher make
or accuracy of tl
to performance,
of any kind caus

Photographs an
file archives and
variety of graph
Internet may be
of text and imag
holders, althoug
17 U.S.C. Sectio

Netscape Comn
Navigator are tr

Netscape Comn
this publication
tions Corporate
tion. All other pr

CONTENTS AT A GLANCE

TABLE OF CONTENTS

INTRO

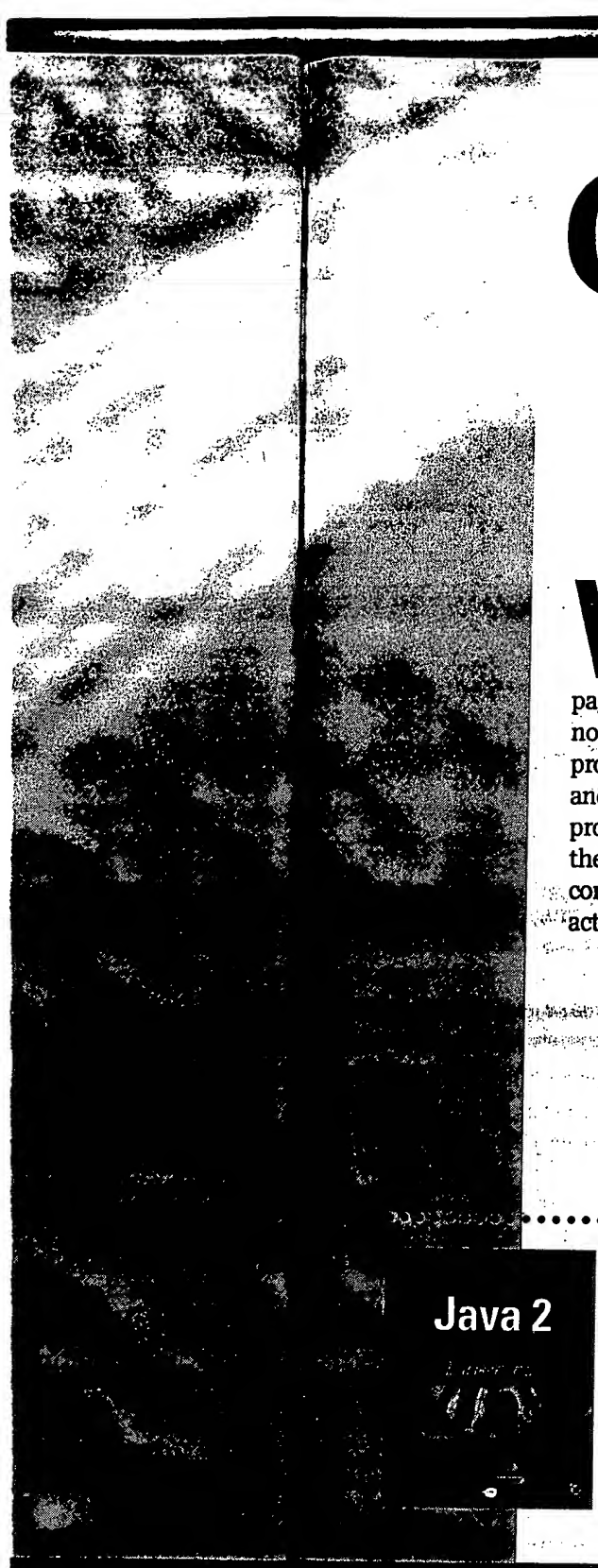
Java 2 (
the bre
compilati
sive cover
This book
goals in n

- Offe
able
- Hel
Java
you
- Acq
styl
thei
you

Java 2
you'll nee
with Java
depths ar

If you'
are many
and hard
piled rep
authors h
the exha
styles. As
which ap
in comm

You'll
authors.
authors
experien
evolutior



Chapter 1

BUILDING THE FIRST JAVA EXAMPLES

Welcome to Java 2! An ambitious agenda lies before you: You're going to get a firm grip on Java programming, creating both powerful Java programs and Web pages, and you will take a guided tour through Java 2. There is no more exciting programming package available. As you are probably aware, the popularity of Java has skyrocketed as more and more people have seen how versatile and powerful it is. Web programmers have found it an excellent tool because it allows them to write programs that will run on many different types of computers. They have started using it to make their Web pages actually *do* something.

Adapted from *Java 2: In Record Time* by Steven Holzner
ISBN 0-7821-2171-3 560 pages $29.99

With Java, you will be able to display animation and images, accept mouse clicks and text, use controls like scrollbars and check boxes, print graphics, support pop-up menus, and even support additional windows and menu bars.

We'll start working on your Java skills right away—you won't need to wade through chapters of abstractions first. We will concentrate on examples, on seeing things from the programmer's point of view—on seeing Java at *work*.

Java programs come two ways: as stand-alone applications and as small programs you can embed in Web pages, called *applets*. Of the two, applets are the most popular, and we'll concentrate primarily on them.

BUILDING THE HELLO EXAMPLE

The first example will be a simple one because right now we just want to get you started in Java without too many extra details to weigh you down. You will create a small Java applet, the type of Java program you can embed in a Web page, that will display the words "Hello from Java!"

What's an Applet?

Just what do I mean by an applet? An applet is a special program that you can embed in a Web page such that the applet gains control over a certain part of the Web page. On that part of the page, the applet can display buttons, list boxes, images, and more. Applets make Web pages "come alive."

Each applet is given the amount of space (usually measured in pixels) that it requests in a Web page, such as the amount of space shown in Figure 1.1. (Soon I'll show you how an applet "requests" space.) This is the space that the applet will use for its display. We'll place the words "Hello from Java!" in the applet, as shown in Figure 1.2.

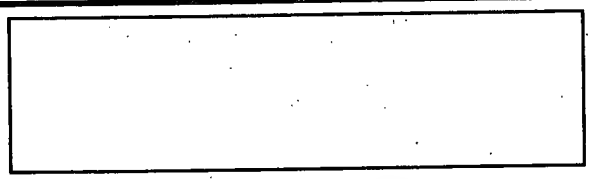


FIGURE 1.1: An applet requests space in a Web page.

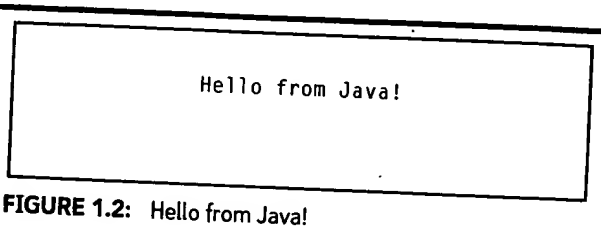


FIGURE 1.2: Hello from Java!

That's how this applet will work; after you create it, you will be able to embed it in a Web page. Let's create and run the applet now.

Creating the Hello Example

Let's call this first applet *hello*. You will store the actual Java code (the lines of text that make up the program) for this applet in a text file named `hello.java`. You'll need an editor of some kind to create this file (such as Windows WordPad or Notepad). You will be creating `.java` files throughout the book, so use an editor you are comfortable with. Also notice that, if you are going to use a word processor like Microsoft Word, you'll have to save your `.java` files as straight text—something you can type out at the DOS prompt and read directly. Check your word processor's Save As menu item or your word processor's documentation to see how to do this. The Sun Java system won't be able to handle anything but straight text files. Now, type the following text into the file `hello.java` (this is the traditional first program in most Java books):

```
import java.awt.Graphics;
public class hello extends java.applet.Applet
{
  public void paint(Graphics g)
  {
    g.drawString("Hello from Java!", 60, 30 );
  }
}

This is the text of your first Java program, and soon you'll
see what each line means. Having typed in the text, save it
to disk as hello.java.
```

NOTE

Note that case counts here—make sure you type `hello.java`, not `Hello.java` or `hello.Java`.

In general, the name of the file will match exactly (including case) the name given in the "class" statement in the file; in this case, that is `hello`:

```
import java.awt.Graphics;

public class hello extends java.applet.Applet
{
  public void paint( Graphics g )
  {
    g.drawString("Hello from Java!", 60, 30 );
  }
}
```

In this book, you will place your programs into subdirectories of a new directory called `java1-2` (this is optional—you can choose any name). That means you'll save the `hello.java` file as `c:\java1-2\hello\hello.java`.

Now you have created `hello.java`. This is the source code for your applet, and it contains the Java code that you have written. The next step is to compile this Java code into a working applet and see your applet at work. Applets have the extension `.class`, making the name of your actual applet `hello.class`. I'll show you *why* applets have the extension `.class` shortly.

SETTING UP THE JAVA JDK

Now you'll use Java itself to create your applet, `hello.class`, from the code `hello.java`. If you haven't already done so, you should install the Java Development Kit (JDK) 1.2.

With previous versions of Java, you used to have to go through a rather lengthy and involved installation process, but that's all changed now—you just have to run an `.EXE` file. You get this `.EXE` file online, from `http://java.sun.com/products/jdk/1.2/`—just download it and follow the instructions for installation.

The next step is to make sure you can run the JDK from any location in your computer (including the `c:\java1-2` directory and its subdirectories, which is where you'll put your Java programs). To do that, make sure the PATH statement in your AUTOEXEC.BAT file (found in the main directory of the C: drive) includes the JDK BIN and LIB directories (here I have installed the JDK in `c:\jdk12`—use whatever path is appropriate to the way you have installed the JDK):

```
PATH=C:\WINDOWS;C:\JDK12\BIN;C:\JDK12\LIB
```

For Windows NT, the path will need to be entered into Start ➢ Settings ➢ Control Panel ➢ System. In the System Properties window, select the Environment tab and set the PATH variable.

The JDK 1.2 is ready to go.

TIP

If you need more help installing the JDK, check out the Troubleshooting Web page at `http://www.javasoft.com`.

You can copy the Java documentation from JavaSoft to the same directory—for example, `c:\JDK12`. Unzip the documentation `.zip` file, creating a `docs` subdirectory (your unzipping program must be able to handle long filenames).

NOTE

You'll need a Web browser to look at the Java documentation because it's formatted in HTML.

Now that you've installed Java 2, let's take a look at what's new in this version of Java.

What's New in 2?

If you're familiar with Java 1.0 or Java 1.1, then you'd probably expect there to be some changes in Java 2, and you'd be right. Let's get an overview of the changes in this new edition of Java. If you're not familiar with Java, you should probably skip to the next section and take a look at this material later—much of this won't make any sense unless you've programmed in Java before.

From Java 1.0 to Java 1.1

Many readers will be familiar with Java 1.0, not Java 1.1, so we will start by looking at the changes from Java 1.0 to Java 1.1.

Abstract Windowing Toolkit enhancements Java 1.1 supports printing, faster scrolling, pop-up menus, the clipboard, a delegation-based event model, imaging and graphics enhancements, and more. In addition, it's faster than Java 1.0 (something Java programmers can definitely appreciate)!

.jar files .jar (Java Archive) files were introduced in Java 1.1 and let you package a number of files together, zipping them to shrink them, so the user can download many files at once. You can put many applets and the data they need together into one .jar file, making downloading much faster. These files are analogous to .zip files except that your browser will download them and unzip them on-the-fly for you.

Internationalization Java 1.1 lets you develop *locale-specific applets*, including using Unicode characters, a locale mechanism, localized message support, locale-sensitive date, time, time zone, number handling, and more.

Signed applets and digital signatures Java 1.1 can create digitally signed Java applications. A digital signature gives your users a "path" back to you in case something goes wrong. This is one of the new security precautions popular on the World Wide Web.

Remote method invocation In Java 1.1, RMI lets Java objects have their methods invoked from Java code running in other Java sessions. This is sort of similar to Local Remote Procedure Calls (LRPCs).

Object serialization Serialization was new in Java 1.1, and it lets you store objects and handle them with binary input/output streams. Besides allowing you to store copies of the objects you serialize, serialization is also the basis of communication between objects engaged in RMI. Object serialization is similar to MFC Serialization, for those who are familiar with Microsoft's Foundation Classes.

Reflection In Java 1.1, reflection lets Java code examine information about the methods and constructors of loaded classes and make use of those reflected methods and constructors.

Inner classes Java 1.1 makes it easier to create adapter classes. An adapter class is a class that implements an interface required by an API (Applications Programming Interface). An adapter class "delegates" control back to an enclosing main object.

New Java native method interface Native code is code that is written specifically for a particular machine. In Java 1.1, this interface was introduced to provide a standard programming

interface for writing Java native methods. The primary goal is binary compatibility of native method libraries across all Java virtual machine implementations on a given platform. Writing and calling native code can significantly improve execution speeds. Java 1.1 included a powerful new Java native method interface.

`Byte`, `Short`, and `Void` classes In Java 1.1, Byte and Short values can be handled as "wrapped" numbers when you use the new Java classes `Byte` and `Short`. The new `Void` class is a placeholder class that we can derive classes from, rather than use directly.

Deprecated methods Quite a number of Java 1.0 methods were considered obsolete in Java 1.1, and they are marked as deprecated in the Java 1.1 documentation. (The Java compiler now displays a warning when it compiles code that uses a deprecated feature.)

Networking enhancements Networking enhancements in Java 1.1 included support for selected BSD-style socket options in the `java.net` base classes. With Java 1.1, Socket and ServerSocket are non-final, extendable classes. New subclasses of SocketException were added for finer granularity in reporting and handling network errors.

I/O enhancements In Java 1.1, the I/O package was extended with character streams, which are like byte streams except that they contain 16-bit Unicode characters rather than eight-bit bytes. Character streams make it easy to write programs that are independent of a specific character encoding and are therefore easier to internationalize. Nearly all of the functionality available for byte streams is also available for character streams.

That completes this overview of what's new in Java 1.1—if you have no idea what I'm talking about, don't worry, it'll become clear later.

From Java 1.1 to Java 2

Now let's have a look at what's new in Java 2.

Security enhancements When code is loaded, it is assigned permissions based on the security policy currently in effect. Each permission specifies a permitted access to a particular

resource (such as "read" and "write" access to a specified file or directory, "connect" access to a given host and port, and so on). The policy, specifying which permissions are available for code from various signers/locations, can be initialized from an external configurable policy file. Unless a permission is explicitly granted to code, it cannot access the resource that is guarded by that permission.

Swing (JFC) Swing is the part of the Java Foundation Classes (JFC) that implements a new set of GUI components with a "pluggable" look and feel. Swing is implemented in pure Java, and is based on the JDK 1.1 Light-weight UI Framework. The pluggable look and feel lets you design a single set of GUI components that can automatically have the look and feel of any platform (e.g., Windows, Solaris, Macintosh).

Java 2D (JFC) The Java 2D API is a set of classes for advanced 2D graphics and imaging. It encompasses line art, text, and images in a single comprehensive model.

Accessibility (JFC) Through the Java Accessibility API, developers will be able to create Java applications that can interact with assistive technologies such as screen readers, speech recognition systems, and Braille terminals.

Drag and Drop (JFC) Drag and Drop enables data transfer across both Java and native applications, between Java applications, and within a single Java application.

Collections The Java Collections API is a unified framework for representing and manipulating Java collections (I'll show you more about them later), allowing them to be manipulated independent of the details of their representation.

Java extensions Framework Extensions are packages of Java classes (and any associated native code) that application developers can use to extend the core platform. The extension mechanism allows the Java Virtual Machine (JVM) to use the extension classes in much the same way it uses the system classes.

JavaBeans enhancements Java 2 provides developers with standard means to create more sophisticated JavaBeans components and applications that offer their customers more seamless integration with the rest of their runtime environment,

such as the desktop of the underlying operating system or the browser.

Input method framework The input method framework enables all text-editing components to receive Japanese, Chinese, or Korean text input through standard input methods.

Package version identification "Versioning" introduces package level version control where applications and applets can identify (at runtime) the version of a specific Java Runtime Environment, VM, and class package.

RMI enhancements Remote Method Invocation (RMI) has several new enhancements including Remote Object Activation, which introduces support for remote objects and automatic object activation, as well as Custom Socket Types that allow a remote object to specify the custom socket type that RMI will use for remote calls to that object. (RMI over a secure transport, such as SSL, can be supported using custom socket types.)

Serialization enhancements Serialization now includes an API that allows the serialized data of an object to be specified independently of the fields of the class. This allows serialized data fields to be written to and read from the stream using the existing techniques (this ensures compatibility with the default writing and reading mechanisms).

Reference objects A reference object encapsulates a reference to some other object so that the reference itself may be examined and manipulated like any other object. Reference objects allow a program to maintain a reference to an object that does not prevent the object from being reclaimed by the Java "garbage collector," which manages memory.

Audio enhancements Audio enhancements include a new sound engine and support for audio in applications as well as applets.

Java IDL Java IDL adds CORBA (Common Object Request Broker Architecture) capability to Java, providing standards-based interoperability and connectivity. Java IDL enables distributed Web-enabled Java applications to invoke operations transparently on remote network services using the industry

standard OMG IDL (Object Management Group Interface Definition Language) and IIOP (Internet Inter-ORB Protocol) defined by the Object Management Group.

JAR enhancements The enhancements include added functionality for the command-line JAR tool for creating and updating signed JAR files. There are also new standard APIs for reading and writing JAR files.

JNI enhancements The Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the Java Virtual Machine into native applications. The primary goal is binary compatibility of native method libraries across all Java Virtual Machine implementations on a given platform. Java 2 extends the Java Native Interface to incorporate new features in the Java platform.

JVMDI A new debugger interface, the Java Virtual Machine, now provides low-level services for debugging. The interface for these services is the Java Virtual Machine Debugger Interface (JVMDI).

JDBC enhancements Java Database Connectivity (JDBC) is a standard SQL database access interface, providing uniform access to a wide range of relational databases. JDBC also provides a common base on which higher-level tools and interfaces can be built. The Java 2 software bundle includes JDBC and the JDBC-ODBC bridge.

These concepts will become clearer as we proceed. Now, you're ready to compile the hello applet and see it at work.

Compiling the Hello Applet

Now that you have installed the JDK and have your `hello.java` source file ready to go, you can create the actual applet and see it run. To do this, change to the `c:\java1-2\hello` directory now (or wherever you have saved the `hello.java` file); this is how the DOS prompt should look:

```
c:\java1-2\hello>
```

Next, type this to create your applet:

```
c:\java1-2\hello>javac hello.java
```

The name of the Java program that takes your Java code and turns it into `.class` files ready to run in Web pages is `javac.exe`, the Java *compiler* (i.e., it compiles `.java` files into `.class` files). If you type the DOS command `Dir` to look at the current directory contents, you should see both `hello.java` and `hello.class`. Because you've created `hello.class`, your applet is ready to go—but what does that mean? What have you really done?

UNDERSTANDING JAVA

Let's take the time now to get an overview of Java. As in most programming languages, we write Java code using words and numbers that are then translated—that is, *compiled*—into binary files that computers can understand. The `hello.java` program is an example of this—you write it such that you can understand it, but when you want to actually run your program, you have to compile it into something a computer can use. In this case, that means using the Java compiler to produce the file named `hello.class`. `hello.class` is a binary file of *bytecodes* that Java-compatible Web browsers can run to produce the desired result. In this way, several lines of Java program code can be compiled neatly into a few bytes. Those bytes are what is actually downloaded when Web browsers read the Web page in which you have placed your Java applets—that is to say, the actual applet is a `.class` file, like `hello.class`, and those are the files you place on your Internet Service Provider's server so other people's Web browsers can download them, as you'll see very shortly.

Experienced programmers may wonder about these bytecodes—why isn't Java simply compiled into the normal machine code that each computer really runs? Because Java bytecodes were intentionally made machine-independent so that they could be run on a wide variety of machines, and that is what originally made them so popular on the Internet—it doesn't matter what type of machine you're downloading to, as long as the user's Web browser can run Java. The downloaded bytecodes are run by the *Java Virtual Machine*, or JVM, and it is the JVM's task to convert bytecodes into the machine language that users' individual computers can run.

The JVM is actually a hypothetical chip that runs Java—it is almost always software, not hardware, that runs Java. Each Web browser that supports Java has a JVM built right into it, and it loads the `.class` file

that makes up your applet with JVM's *class loader* and then runs the applet.

Running the Hello Applet

To see `hello.class`, your first applet, running, you'll need a Web page to place it in. Use your editor again to create a new file, `hello.htm`, which will be your Web page, written in the language of Web pages, *HyperText Markup Language* (HTML) (we'll review HTML in a minute). Enter the following text into `hello.htm` and save it in the same directory as the `hello.class` file:

```
<html>

<!- Web page written for the Sun Applet Viewer>

<head>
<title>hello</title>
</head>

<body>
<hr>

<applet
code=hello.class
width=200
height=200>

</applet>

<hr>
</body>
</html>
```

Now you can run the hello applet by simply viewing this new Web page, `hello.htm`. To do that, use the Applet Viewer that comes with the JDK 1.2. To use the Applet Viewer, go back to the `hello` subdirectory and type the following:

```
c:\java1-2\hello>appletviewer hello.htm
```

Again, capitalization is very important here—make sure your capitalization matches the exact spelling of the Web page name. When you've done

this, the Applet Viewer runs, as shown below—and you see your message, "Hello from Java!" Your first applet is a success.

TIP

You can use any Java-enabled Web browser to look at this Web page. For most of the applets in this book, however, you will have to use either a Web browser that supports Java 2 (not just Java 1.0 or Java 1.1) or the Sun Applet Viewer.

Your first applet, `hello.class`, runs—but what exactly did you do? Let's take a look now at the Java code that you entered for `hello.java`, examining it line by line to get a better idea of how Java programming works (even though Java will handle many of these details for you later).

Understanding the Hello Example

Let's take apart your first applet now. Begin with this line:

```
import java.awt.Graphics;
```

What does this mean? This line actually points out one of the great advantages of Java programming. When you're adding menus and separate windows to your Java applets, you can imagine that it would be a great deal of work to create everything from scratch—that is, write the entire code for menu handling, separate window creation, and so forth. Instead of asking you to do so, Java comes complete with several predefined libraries, and much of this book will be an examination of the routines in these libraries. You'll learn more about this later, but what you're doing is adding support from the main Java graphics library of routines

to your applet. In this way, we'll be able to draw the text string, "Hello from Java!", in the applet's window.

NOTE

If you're a C/C++ programmer, you'll notice that the import statement works much like the C/C++ #include statement.

Next, add these lines to `hello.java`:

```
import java.awt.Graphics;

public class hello extends java.applet.Applet

{
    .
    .
    .
```

You've just created a Java *class* named `hello`. What does this mean?

OBJECT-ORIENTED PROGRAMMING

Objects and *classes* are two fundamental concepts in object-oriented languages like Java. There's been a lot of hype about object-oriented programming (OOP), and that can make the whole topic seem mysterious and unapproachable. In fact, object-oriented programming was introduced to make longer programs *easier* to create. We'll start a mini-survey of object-oriented programming by looking at objects.

Understanding Java Objects

In long, involved programs, there can be a profusion of both variables and functions, sometimes hundreds of each. Creating and maintaining the program code can become a very cluttered task because you have to keep so many things in mind. There may also be unwanted interaction if various functions use variables of the same name. Object-oriented programming was invented to break up such large programs.

The idea behind objects is quite simple—you just break up your program into the various parts, each of which you can easily conceptualize as performing a discrete task, and those are your objects. For example, you may put all the screen-handling parts of a program together into

an object named `screen`. Objects are more powerful than simple functions or sets of variables because an object can hold both functions and variables wrapped up together in a way that makes it easy to use. The `screen` object may hold not only all the data displayed on the screen, but also the functions needed to handle that data, like `drawString()` or `drawLine()`. This means that all the screen handling is hidden from the rest of the program in a convenient way, making the rest of the program easier to handle.

As another example, think of a refrigerator. A refrigerator would be far less useful if you had to regulate all the temperatures and pumps and so forth by hand at all times. Making all those functions internal and automatic to the refrigerator makes it into an easy object to deal with and a useful one: a *refrigerator*. Wrapping up code and data into objects this way is the basis of object-oriented programming.

What's a Java Class?

But how do you create objects? That's where *classes* come in. A class is to an object what a cookie cutter is to a cookie—a template or blueprint. In terms of programming, you might think of the relationship between a data type, like an integer, and the actual variable itself like this, where you set up an integer named `the_data`:

```
int the_data;
```

This is the actual way to create an integer variable in Java. Here, `int` is the type of variable you are declaring and `the_data` is the variable itself. This is the same relationship that a class has to an object, and informally you may think of a class as an object's *type*.

TIP

Java supports all the standard C and C++ primitive data types like `int`, `double`, `long`, `float`, and so forth.

For example, if you had set up a class named, say, `graphicsClass`, you can create an object of that class named `screen` this way:

```
graphicsClass screen;
```

You'll see how to actually create a class soon (creating a class like `graphicsClass` is not hard—when you create a class in code, you will just group all its functions and data inside the class definition), and then

you'll see how to create objects of that class. What's important to remember is this: the object itself is what holds the data you want to work with; the class itself holds no data but just describes how the object should be set up.

Object-oriented programming at root is nothing more than a way of grouping functions and the data they work on together to make your program less cluttered. You'll see more about object-oriented programming throughout this book, including how to create a class, how to create an object of that class, and how to reach the functions and data in that object when you want to.

That completes the mini-overview of classes and objects. As you can see, a class is just a programming construct that groups together, or *encapsulates*, functions and data, and an object may be thought of as a variable of that class's type, as the object `screen` is to the class `screenclass`.

As it turns out, Java comes complete with several libraries of predefined classes, which save you a great deal of work. Throughout this book, we will examine these predefined and very useful Java classes. Using these predefined classes, we'll create objects needed to handle buttons, text fields, scroll bars, and much more.

Learning about Java Packages

These class libraries are called *packages* in Java, and one such library is called `java.awt` (where `awt` stands for Abstract Window Toolkit). This library holds the `Graphics` class, which will handle the graphics work you undertake. So this line in the `hello.java` file:

```
import java.awt.Graphics;
```

actually means that you want to include the Java Graphics class and make use of it in your program. In a minute, you will use an object of the `Graphics` class for your graphics output.

You've added support for graphics handling by including the `java.awt.Graphics` class (and in Java, displaying the text string "Hello from Java!" is considered graphics handling). Next, it's time to set up your hello applet itself. To do so, define a new class named `hello`. This is the standard way of setting up an applet in Java, and in fact, the applet itself has the file extension `.class`. That's because each class defined in a `.java` file ends up being exported to a `.class` file, where you can make use of it. You'll learn more details about this soon.

It would be quite difficult to write all the code an applet class needs from scratch. For example, we'd need to interact with the Web browser, reserve a section of screen, initialize the appropriate Java packages, and much more. It turns out that all that functionality is already built into the Java `Applet` class, which is part of the `java.applet` package. But how do you make use of the `Applet` class? You want to customize the applet to display your text string, and the `java.applet.Applet` class itself knows nothing about that.

Understanding Java Inheritance

You can customize the `java.applet.Applet` class by *deriving* the `hello` class from the `java.applet.Applet` class. This makes `java.applet.Applet` the *base* class of the `hello` class, and it makes `hello` a class derived from `java.applet.Applet`. This gives you all the power of the `java.applet.Applet` class without the worries of writing it yourself, and you can add what you want to this class by adding code to your derived class `hello`.

This is an important part of object-oriented programming, and it's called *inheritance*. In this way, a derived class inherits the functionality of its base class and adds more on top of it. For example, you may have a base class called `chassis`. You can derive various classes from this base class called, say, `car` and `truck`. In this way, two derived classes can share the same base class, saving time and effort programmatically. Although the `car` and `truck` classes share the same base class, `chassis`, they added different items to the base class, ending up as two quite different classes, `car` and `truck`.

Using inheritance, then, you will *extend* the base class `java.applet.Applet` by creating your own class `hello` and adding onto the base class. In the `hello.java` source, you indicate that the `hello` class is derived from the `java.applet.Applet` class like this (note that you use the keyword *class* to indicate that you are defining a new class):

```
import java.awt.Graphics;

public class hello extends java.applet.Applet
{
    .
    .
    .
```

In starting to set up the new class, `hello`, you've given it all the power of the `java.applet.Applet` class (like the ability to request space from the Web browser and to respond to many browser-created commands). But how do you make additions and even alterations to the `java.applet.Applet` class to customize your own `hello` class? How do you display your text string? One way is by *overriding* the base class's built-in functions (overriding is an important part of object-oriented programming). When you redefine a base class's function in a derived class, the new version of the function is the one that takes over. In this way, you can customize the functions from the base class as you like them in the derived class.

For example, one function in the `java.applet.Applet` class is called `paint()`. This is a very important function that is called when the Web browser tells the applet to create its display on the screen. This happens when the applet first begins and every time it has to be redisplayed later (for example, if the Web browser was minimized and then maximized, or if some window was moved and the applet's display area was uncovered after having been covered).

Your goal in the `hello` class is to display the string "Hello from Java!" on the screen, and in fact, you will override the `java.applet.Applet` class's `paint()` function to do so. You override a base class's function simply by redefining it in the new class. Do that now for the `paint()` function, noting first that the built-in functions of a class are called that class's *methods*. In this case, then, you override (that is, redefine) the `paint()` method like this:

```
import java.awt.Graphics;

public class hello extends java.applet.Applet
{
  public void paint( Graphics g )[
  {
    .
    .
    .
```

NOTE

The built-in functions of a class are called *methods*. Classes can also have built-in variables—called *data members*—and even constants. Collectively, all these parts are called a class's *members*.

What Are Java Access Modifiers?

The keyword *public* is called an *access modifier*. A class's methods can be declared `public`, `private`, or `protected`. If they are declared `public`, then you can call them from anywhere in the program, not just in the class in which they are defined. If they are `private`, they may be called from only the class in which they are defined. If they are `protected`, they may be called from only the class in which they are defined and the classes derived from that class.

Next, indicate the *return* type of the `paint()` method. When you call a method, you can pass parameters to it, and it can return data to you. In this case, `paint()` has no return value, which you indicate with the return type *void*. Other return types are `int` for an integer return value (this variable is usually 32 bits long), `long` for a long integer (this variable is usually 64 bits long), `float` for a floating point return, or `double` for a double-precision floating point value. You can also return arrays and objects in Java.

Finally, note that you indicate that the `paint()` method is automatically passed one parameter—an object of the `Graphics` class called `g`:

```
import java.awt.Graphics;

public class hello extends java.applet.Applet
{
  public void paint(Graphics g)
  {
    .
    .
    .
```

This Graphics object represents the physical display of the applet. That is, you can use the built-in methods of this object—such as `drawImage()`, `drawLine()`, `drawOval()`, and others—to drawn on the screen. In this case, you want to place the string "Hello from Java!" on the screen, and you can do that with the `drawString()` method.

How do you reach the methods of an object like the `Graphics` object named `g`? You do that with a dot operator (`.`) like this: `g.drawString()`, where here you are invoking `g`'s `drawString()` method to "draw" a string of text on the screen (text is handled like any other type of graphics in a windows environment—that is, it is drawn on the screen rather than "printed," just as you would draw a rectangle or circle). Supply three parameters to the `drawString()` method—the string of text you want to display, and the (x, y) location of that string's lower-left corner (called the

starting point of the string's *baseline*) in pixels on the screen, passed in two integer values. As shown in Figure 1.3, you can draw your string at the pixel location (60, 30), where (0, 0) is the upper-left corner of the applet's display.

NOTE

The coordinate system in a Java program is set up with the origin (∅, ∅) at the upper left, with x increasing horizontally to the right and y increasing vertically downwards; this fact will be important throughout the book. If it seems backwards to you, you might try thinking of it in terms of reading a page of text, like this one, where you start at the upper-left and work your way to the right and down. The units of measurement in Java coordinate systems are almost always screen pixels.

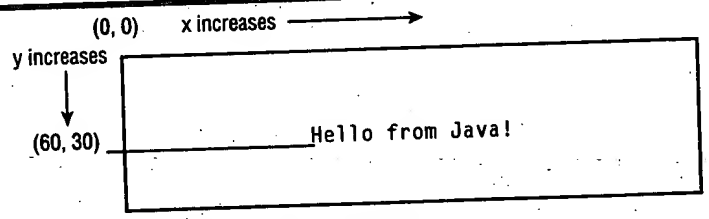


FIGURE 1.3: Drawing a string at (60,30)

This means that you add a call to the `drawString()` method this way:

```
import java.awt.Graphics;
public class hello extends java.applet.Applet
{
  public void paint( Graphics g )
  {
    g.drawString( "Hello from Java!", 60, 30 );
  }
}
```

Note that Java uses the same convention as C or C++ to indicate that a code statement is finished: it ends the statement with a semicolon (;).

TIP

In general, Java adheres very strongly to C++ coding conventions. If you know C++, you already know a great deal of Java.

You have completed the code necessary for this applet, which is also to say you have completed the code for the new class, `hello`. When the Java compiler creates `hello.class`, the entire specification of the new class will be in that file. This is the actual binary file that you upload to your Internet Service Provider so that it may be included in your Web page. A Java-enabled Web browser takes this class specification and creates an object of that class and then gives it control to display itself and, if applicable, handle user input.

But how? You have not yet completed the dissection of the first example; all you have done so far is to trace the development of `hello.java` into `hello.class`. How did you get the applet to be displayed in the Applet Viewer?

Understanding the Applet's Web Page

The Applet Viewer took the `hello.class` applet and displayed it in a Web page, as shown in Figure 1.4.

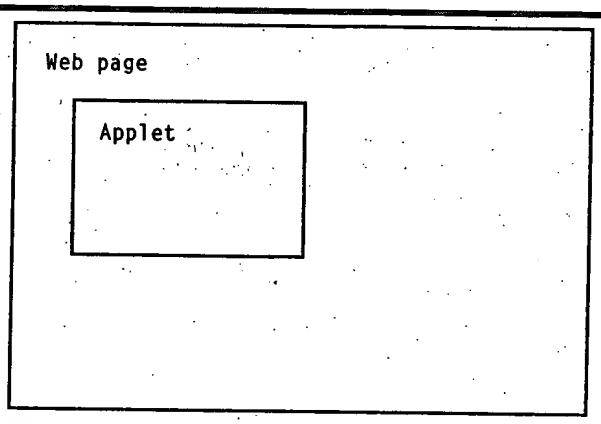


FIGURE 1.4: Displaying an applet in a Web page

How did it get there? You created a Web page for your applet and then opened that Web page in the Applet Viewer, which then displayed your applet. That Web page looks like this:

```
<html>

<!- Web page written for the Sun Applet Viewer>

<head>
```

```
<title>hello</title>
</head>

<body>
<hr>

<applet
code=hello.class
width=200
height=200>

</applet>

<hr>
</body>
</html>
```

Web pages are written in HTML (HyperText Markup Language). Because applets appear in Web pages, we will take the time to briefly work through the above page to make sure you know what's going on. If you're familiar with HTML, you can skip much of this review, but you should take a look at how to use the `<applet>` tag to embed applets in Web pages.

CONNECTING JAVA AND HTML

Let's take apart the Web page you created for the applet now, starting with the `<html>` tag:

```
<html>
.
.
.
```

Instructions in `.html` pages are placed into tags surrounded by angle brackets: < and >. The tags hold directions to the Web browser and are not displayed on the screen. Here, the `<html>` tag indicates to the Web browser that this `.html` file is written in HTML.

Next comes a comment. Comments in `.html` pages are written using the ! symbol like this: `<! This is a comment.>`. Indicate that this is a Web page written so that we can use the Sun Applet Viewer, like this:

```
<html>

<!- Web page written for the Sun Applet Viewer>
```

Next comes the header portion of the Web page, which you declare with the <head> tag, ending the header section with the corresponding end header tag, </head> (many HTML tags are used in pairs like this, such as <head> and </head>, or and to center text and images). In this case, the .html file gets the title (set up with the <title> tag) *hello*, to match your applet:

```
<html>

<!- Web page written for the Sun Applet Viewer>

<head>
<title>hello</title>
</head>
```

The title is the name given to a Web page, and it's usually displayed in the Web browser's title bar. Next comes the body of the Web page. Here is where all the actual items for display will go. You start the page off with a ruler line (visible in Figure 1.4), using the <hr> tag:

```
<html>

<!- Web page written for the Sun Applet Viewer>

<head>
<title>hello</title>
</head>

<body>
<hr>
```

Now we come to the applet. Applets are embedded with the <applet> tag, and here you use the *code* keyword to indicate that this applet is supported by the hello.class file. You indicate the size of the applet as 200 × 200 pixels (you can choose any size you like here) this way:

```
<html>

<!- Web page written for the Sun Applet Viewer>
```

```
<head>
<title>hello</title>
</head>

<body>
<hr>
<applet
code=hello.class
width=200
height=200>
</applet>
  .
  .
  .
```

TIP

You can also use the `java.applet.Applet.resize()` method in your source code to request that the Web browser resize applets.

The `<applet>` tag is important, so let's take a closer look at it now. Here's how the `<applet>` tag works in general (the items in square brackets are optional, and the others are required):

```
<APPLET>
  [ALIGN = LEFT or RIGHT or TOP or TEXTTOP or MIDDLE or
    ABSMIDDLE or BASELINE or BOTTOM or ABSBOTTOM]
  [ALT = AlternateText]
  CODE = AppletName.class
  [CODEBASE = URL of .class file]
  HEIGHT = AppletPixelsHeight
  [HSPACE = PixelSpaceToLeftOfApplet]
  [NAME = AppletInstanceName]
  [VSPACE = PixelSpaceAboveApplet]
  WIDTH = AppletPixelsWidth
  >
  [<PARAM NAME = Parameter1 VALUE = VALUE1]
  [<PARAM NAME = Parameter2 VALUE = VALUE2]
        .
        .
        .
</APPLET>
```

TIP

You can specify the URL of the applet's `.class` file with the CODEBASE keyword. This is often useful if you want to store your applets together in a directory in your ISP, away from the `.html` files.

Indicate to the Web browser here how much space you'll need for your applet, using the `HEIGHT` and `WIDTH` keywords. You can also pass parameters to applets with the PARAM keyword like this: `<applet> PARAM today = "friday" </applet>`. Passing parameters in this way allows you to customize your applets to fit different Web pages because you can read the parameters from inside an applet and make use of them.

TIP

There are enhancements to the `<applet>` tag in Java 2, such as the ability to pass the name of `.jar` files as parameters. You'll learn more about this later on.

Not all Web browsers support Java. In practice, this means that those browsers just ignore the `<applet>` tag. This, in turn, means that you can place text between the `<applet>` and `</applet>` tags that will be displayed in non-Java browsers (and not in Java-enabled browsers), like this:

```
<applet code=hello>
Your Web browser does not support Java, so you can't see my
applets, sorry!
</applet>
```

Using the `<applet>` tag, you can embed applets in Web pages, as Java has done in this temporary page. Finish off the Web page with the `</body>` and `</html>` tags as follows:

```
<html>

<!- Web page written for the Sun Applet Viewer>

<head>
<title>hello</title>
</head>

<body>
<hr>

<applet
code=hello.class
width=200
```

```
height=200>

</applet>

<hr>
</body>
</html>
```

This completes our first example—you've had a glimpse into the process of creating and running an applet. It was as quick and easy as that—you created and ran your first applet.

What's Next?

In this chapter, the example applet demonstrated the easiest way to get an applet to work. Let's continue on to get a better idea of how you'll be working with Java throughout the book as you give your applet more power in Chapter 2.

Java 2 COMPLETE

Java 2 Complete [illegible]

With **Java 2 Complete** [illegible]

Java 2 Complete [illegible]

USER LEVEL	ALL LEVELS
BOOK TYPE	HOW-TO/REFERENCE
CATEGORY	INTERNET PROGRAMMING

SYBEX®

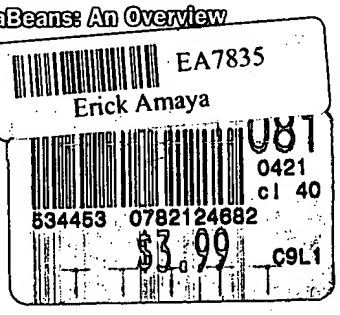
Inside:

- Building the First Java Examples
- Handling Java Text Fields
- Using Java Buttons
- Using Java Layouts and Check Boxes
- Working with Radio Buttons
- Adding Scroll Bars

- Applets, Applications, and the Java Development Kit
- Working With Java Objects
- Exception Handling
- Standard Java Packages
- File I/O and Streams

- Custom Components
- The JFC Swing Components
- Threads and Multithreading
- Java Database Connectivity (JDBC)
- The 2D API

- JavaBeans: An Overview
- Be[illegible]
- The[illegible]



A New Public Key Cryptosystem Based on Higher Residues

David Naccache
Gemplus Card International
34 rue Guynemer
Issy-les-Moulineaux CEDEX, 92447, France
naccache@compuserve.com

Jacques Stern
Ecole Normale Supérieure
45 rue d'Ulm
Paris CEDEX 5, 75230, France
jacques.stern@ens.fr

Abstract

This paper describes a new public-key cryptosystem based on the hardness of computing higher residues modulo a composite RSA integer. We introduce two versions of our scheme, one deterministic and the other probabilistic. The deterministic version is practically oriented: encryption amounts to a single exponentiation w.r.t. a modulus with at least 768 bits and a 160-bit exponent. Decryption can be suitably optimized so as to become less demanding than a couple RSA decryptions. Although slower than RSA, the new scheme is still reasonably competitive and has several specific applications. The probabilistic version exhibits an homomorphic encryption scheme whose expansion rate is much better than previously proposed such systems. Furthermore, it has semantic security, relative to the hardness of computing higher residues for suitable moduli.

1 Introduction

It is striking to observe that two decades after the discovery of public-key cryptography, the cryptographer's toolbox still contains very few asymmetric encryption schemes. Consequently, the search for new public-key mechanisms remains a major challenge. The quest appears sometimes hopeless as new schemes are immediately broken or, if they survive, are compared with RSA, which is obviously elegant, simple and efficient.

Similar investigations have been relatively successful in the related setting of identification, where a user attempts to convince another entity of his identity by means of an on-line communication. For example, there have been several attempts to build identification protocols based on simple operations (see [33, 35, 36, 26]). Although the question of devising new public-key cryptosystems appears much more difficult (since it deals with trapdoor functions rather than simple one-way functions), we feel that research in this direction is still in order: simple yet efficient constructions may have been overlooked.

The scheme that we propose in the present paper uses an RSA integer n which is a product of two primes p and q, as usual. However, it is quite different from RSA in many respects:

1. it encrypts messages by exponentiating them with respect to a fixed base rather than by raising them to a fixed power
2. it uses a different "trapdoor" for decryption
3. its strength is not directly related to the strength of RSA
4. it exhibits further "algebraic" properties that may prove useful in some applications.

We briefly comment on those differences. The first one may offer a competitive advantage in environments where a large amount of memory is available: such environments allow impressive speed-ups in exponentiations that do not have analogous counterparts in RSA-like operations. The second is of obvious interest in view of the fact quoted above that there are very few public-key cryptosystems available. Without going into technical details at this point, let us simply mention that the new trapdoor is obtained by injecting small prime factors in $p-1$ and $q-1$. In order to understand what the third difference is, we note that, if the modulus n can be factored, then both RSA and the proposed cryptosystem are broken. However, it is an open problem whether or not RSA is "equivalent" to factoring, which would mean that breaking RSA allows to factor. For this reason, the hypothesis that RSA is secure has become an assumption of its own, formally stronger than factoring. Our cryptosystem is related to another hypothesis, also formally stronger than factoring and known as the higher residuosity assumption. This may help to understand how these various hypotheses are related. Finally, we will explain the algebraic property of our scheme (called the *homomorphic property*) by means of an example: suppose that one wishes to withdraw a small amount u from the balance m of some account; assume further that the balance is given in encrypted form $E(m)$ and that the clerk performing the operation does not have access to decryption. The cryptosystem that we propose simply solves the problem by computing $E(m)/E(u) \bmod n$, which turns out to be the encryption of the new balance $m-u$.

The ability to perform algebraic operations such as additions or substractions by playing only with the cryptograms has potential applications in several contexts. We quote a few:

1. in election schemes, it provides a tool to obtain the tally without decrypting the individual votes (see [4])

5th Conference on Computer & Communications Security
San Francisco CA USA

2. in the area of watermarking, it allows to add a mark to previously encrypted data (as explained in [25]).

Still, in these contexts, it is often needed to encrypt data taken from a small set S (e.g. 0/1 votes) and it is well known that deterministic cryptosystems, such as RSA, fail here: in order to decrypt $E(a)$, one can simply compare the ciphertext with the encryptions of all members of S and thus find the correct value of a. In order to overcome the difficulty, one has to use probabilistic encryption, where each plaintext has many corresponding ciphertexts, depending on some additional random parameter chosen at encryption time. Such a scheme should make it impossible to distinguish encryptions of distinct values, even if these are restricted to range over a set with only two elements. This very strong requirement has been termed *semantic security* ([12]). As a further difference with RSA, the cryptosystem introduced in this paper, has a very natural probabilistic version, with proven semantic security.

The probabilistic homomorphic encryption schemes proposed so far suffer from a serious drawback: they have very poor bandwith. Typically, they need something like one kilobit to encrypt just a few bits, which is a quite severe expansion rate. This may be acceptable for election schemes but definitely hampers other applications. The main achievement of the present paper is to reach a significant bandwith, while keeping the other properties, including semantic security.

Before we turn to the more technical developments of our paper, it is in order to compare it with earlier work: it is indeed the case that the question of finding trapdoors for the discrete logarithm problem has been the subject of many papers. At this point, it is fair to mention that the probabilistic cryptosystem that we propose is actually quite close to the most general case of the homomorphic encryption schemes introduced by Benaloh in his Ph-D thesis [4]. Still, both in this thesis and in the related work ([5, 6, 7]), the security and potential applications are only investigated in a setting where the bandwith remains small. A more recent paper by Park and Won (see [24])describes a related probabilistic cryptosystem using a trapdoor based on injecting a single power of a small odd integer into $p-1$ or $q-1$ and proves its security with respect to an *ad hoc* statement. Thus, our paper offers the first thorough discussion of the security of a probabilistic homomorphic encryption scheme with significant bandwith. After the completion of the present work, we have been informed that another homomorphic probabilistic encryption scheme, using moduli n of the form p^2q, where p and q are primes, had been found by Okamoto and Uchiyama (see [22]), achieving an expansion rate similar to ours. Finally, it should be emphasized that the deterministic version of our scheme is not simply a twist that fixes the random string in the probabilistic version: considering its practicality, we believe that, even if it is not intended to be a direct competitor to RSA, it enters the very limited list of efficient public-key cryptosystems.

The paper is organized as follows: in the next two sections, we successively describe the deterministic and the probabilistic version of our scheme, the former with a practical approach, the latter in a more complexity-theoretic spirit. We then discuss applications and end up with a challenge for the research community.

2 The deterministic version

As was just mentioned, our approach to the deterministic scheme is practically oriented: we discuss system set-up and key-generation, encryption and decryption, with performances in mind. We also carry on a security analysis at the informal level and we derive minimal sugested parameters.

2.1 System set-up and key generation

The scheme that we propose in the present paper can be described as follows: let σ be a squarefree odd B-smooth integer, where B is small integer and let $n = pq$ be an RSA modulus such that σ divides $\phi(n)$ and is prime to $\phi(n)/\sigma$. Typically, we think of B as being a 10 bit integer and we consider n to be at least 768 bits long. Let g be an element whose multiplicative order modulo n is a large multiple of σ. Publish n, g and keep p, q and optionally σ secret. A message m smaller than σ is encrypted by $g^m \bmod n$; decryption is performed using the prime factors of σ as will be seen in the next subsection.

Generation of the modulus appears rather straightforward: pick a family p_i of k small odd distinct primes, with k even. Set $u = \prod_{i=1}^{k/2} p_i$, $v = \prod_{k/2+1}^{k} p_i$ and $\sigma = uv = \prod_{i=1}^{k} p_i$. Pick two large primes a and b such that both $p = 2au + 1$ and $q = 2bv + 1$ are prime and let $n = pq$.

However, this generation is lengthy especially when the size of the modulus grows: a has to be chosen in the appropriate range and tested for primality as well as $p = 2au + 1$ until both tests succeed simultaneously. This might be a bit time-consuming. Instead, we suggest to generate a, b, u and v first (independently of any primality requirements on p and q) and use a couple of 24-bit "tuning primes" p' and q' (not used in the encryption process) such that $p = 2aup' + 1$ and $q = 2bvq' + 1$ are primes. To avoid interferences with the encryption mechanics, we recommend to make sure that $\gcd(p'q', \sigma) = 1$ and $p' \neq q'$. In practice, such an approach is only 9% slower than equivalent-size RSA key-generation.

To select g, one can choose it at random and check whether or it has order $\phi(n)/4$. The main point is to ensure that g is not a p_i-th power, for each $i \leq k$ by testing that $g^{\frac{\phi(n)}{p_i}} \neq 1 \bmod n$. The success probability is :

$$\pi = \prod_{i=1}^{k}(1 - \frac{1}{p_i}), \text{ whose logarithm is : } \ln(\pi) \simeq -\sum_{i=1}^{k}\frac{1}{p_i}$$

If the p_is are the first k primes, this in turn can be estimated as $-\ln\ln k$ and results in the quite acceptable overall probability of $\pi \cong 1/\ln k$. Another method consists in choosing, for each index $i \leq k$, a random g_i until it is not a p_i-th power. With overwhelming probability $g = \prod_{i=1}^{k} g_i^{\sigma/p_i}$ has order $\geq \phi(n)/4$.

2.2 Encryption and Decryption

Encryption consists in a single modular exponentiation: a message m smaller that σ is encrypted by $g^m \bmod n$. Note that it does not require knowledge of σ. A lower bound (preferably a power of two) is enough but it is unclear how important for the security of the scheme is keeping σ secret. However, if one chooses to keep σ secret, necessary precautions (similar to these applied to Rabin's scheme [31] or Shamir's RSA for paranoids [34]) should be enforced for not being used as an oracle[1].

[1]For example, an attacker having access to a decryption box can decrypt $g^m \bmod n$ for some $m > \sigma$ and get $m \bmod \sigma$. This discloses (by subtraction) a multiple of σ and σ can then be found by a few re-

Also, there is actually no reason why the p_is should be prime. Everything goes through, *mutatis mutandis*, as soon as the p_is are mutually prime. Thus, for example, they can be chosen as prime powers, which is a way to increase the variability of the scheme.

Decryption is based on the chinese remainder theorem. Let p_i, $1 \leq i \leq k$, be the prime factors of σ. The algorithm computes the value m_i of m modulo each p_i and gets the result by chinese remaindering, following an idea which goes back to the Pohlig-Hellman paper [27]. In order to find m_i, given the ciphertext $c = g^m \bmod n$, the algorithm computes $c_i = c^{\frac{\phi(n)}{p_i}} \bmod n$, which is exactly $g^{\frac{m_i\phi(n)}{p_i}} \bmod n$. This follows from the following easy computations, where y_i stands for $\frac{m-m_i}{p_i}$:

$$c_i = c^{\frac{\phi(n)}{p_i}} = g^{\frac{m\phi(n)}{p_i}} = g^{\frac{(m_i+y_ip_i)\phi(n)}{p_i}}$$

$$= g^{\frac{m_i\phi(n)}{p_i}} g^{y_i\phi(n)} = g^{\frac{m_i\phi(n)}{p_i}} \bmod n$$

By comparing this result with all possible powers $g^{\frac{j\phi(n)}{p_i}}$, it finds out the correct value of m_i. In other words, one loops for $j = 0$ to $p_i - 1$ until $c_i = g^{\frac{j\phi(n)}{p_i}} \bmod n$.

The cleartext m can therefore be computed by the following procedure :

```
for i = 1 to k
{
 let c_i = c^{φ(n)/p_i} mod n
 for j = 0 to p_i − 1
 {if c_i == g^{jφ(n)/p_i} mod n let m_i = j}
}
x = ChineseRemainder({m_i}, {p_i})
```

The basic operation used by this (non-optimized) algorithm is a modular exponentiation of complexity $\log^3(n)$, repeated less than :

$$k\, p_k < \log(n)\, p_k \cong \log(n)\, k\, \log(k) < \log^2(n)\, \log\log(n)$$

times. Decryption therefore takes $\log^5(n) \log\log(n)$ bit operations.

This is clearly worse than the $\log^3(n)$ complexity of RSA but encryption can be optimized if a table stores all possible values of $t[i,j] = g^{\frac{j\phi(n)}{p_i}}$, for $1 \leq i \leq k$ and $1 \leq j \leq i$: the value m_i of the cleartext m modulo p_i is found by table look-up, once $c^{\frac{\phi(n)}{p_i}} \bmod n$ has been computed. It is not really necessary to store all $g^{\frac{j\phi(n)}{p_i}}$. Any hash function that distinguishes $g^{\frac{j\phi(n)}{p_i}}$ from $g^{\frac{j'\phi(n)}{p_i}}$, for $j \neq j'$ will do and, in practical terms, a few bytes will be enough, for example approximately $2|p_i|$ bits from each $t[i,j]$. It is even possible to use hash functions that do not discriminate values of $g^{\frac{j\phi(n)}{p_i}}$: the proper one is spotted by considering, by table look-up hashes of $g^{\frac{2^\ell j\phi(n)}{p_i}}$, for $\ell = 1, 2, \cdots$ until there is no ambiguity. This can be very efficiently implemented by storing hash values in increasing order w.r.t. ℓ and one single bit might be enough.

peated trials and gcds. To prevent such an action, the decryption box cannot only re-encrypt and check against the ciphertext received, as this allows a search by dichotomy. It should first check that the cleartext is in the appropriate range, e.g. $< 2^t$ with $2^t < m$, re-encrypt it and then check that it matches up with the original ciphertext before letting anything out.

2.3 A toy example

• **key generation for $k = 6$**
$p\ 21211 = 2 \times 101 \times 3 \times 5 \times 7 + 1$,

$q = 928643 = 2 \times 191 \times 11 \times 13 \times 17 + 1$,

$n = 21211 \times 928643 = 19697446673$ and $g = 131$ yield the table:

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
$j=0$	0001	0001	0001	0001	0001	0001
$j=1$	1966	6544	1967	6273	6043	0372
$j=2$	9560	3339	4968	7876	4792	7757
$j=3$		9400	1765	8720	0262	3397
$j=4$		5479	6701	7994	0136	0702
$j=5$			6488	8651	6291	4586
$j=6$			2782	4691	0677	8135
$j=7$				9489	1890	3902
$j=8$				8537	6878	5930
$j=9$				2312	2571	6399
$j=10$				7707	7180	6592
$j=11$					8291	9771
$j=12$					0678	0609
$j=13$						7337
$j=14$						6892
$j=15$						3370
$j=16$						3489

where entry $\{i,j\}$ contains $g^{j\phi(n)/p_i} \bmod n \bmod 10000$.
• **encryption of $m = 202$**

$$c = g^m \bmod n = 131^{202} \bmod 19697446673 = 519690214$$

• **decryption**
by exponentiation, we retrieve :

$$c^{\frac{\phi(n)}{P1}} \bmod n \bmod 10000 = 1966$$
$$c^{\frac{\phi(n)}{P2}} \bmod n \bmod 10000 = 3339$$
$$c^{\frac{\phi(n)}{P3}} \bmod n \bmod 10000 = 2782$$
$$c^{\frac{\phi(n)}{P4}} \bmod n \bmod 10000 = 7994$$
$$c^{\frac{\phi(n)}{P5}} \bmod n \bmod 10000 = 1890$$
$$c^{\frac{\phi(n)}{P6}} \bmod n \bmod 10000 = 3370$$

wherefrom, by table lookup :

$$m \bmod 3 = \texttt{table}(1966) = 1$$
$$m \bmod 5 = \texttt{table}(3339) = 2$$
$$m \bmod 7 = \texttt{table}(2782) = 6$$
$$m \bmod 11 = \texttt{table}(7994) = 4$$
$$m \bmod 13 = \texttt{table}(1890) = 7$$
$$m \bmod 17 = \texttt{table}(3370) = 15$$

and by Chinese remaindering : $m = 202$.

2.4 Suggested parameters and security analysis

We suggest to take $\sigma > 2^{160}$ and we consider $|n| = 768$ bits as a minimum size for the modulus.

If the factorization of n is found, then a and b become known as well as $\phi(n)$. The scheme is therefore broken. However, the scheme does not appear to be provably equivalent to factoring. Rather, it is related to the question of having oracles that decide whether or not a random number x is a p_i-th power modulo n, for $i = 1, \ldots, k$. This is known as the higher residuosity problem and is currently considered unfeasible. Formal equivalence of this problem and the probabilistic version of our encryption scheme will be proved in the next session. Considering the basic deterministic version, we have no formal proof but we haven't found any plausible line of attack either. Also, the efficient factoring methods such as the quadratic sieve (QS) or the number field sieve (NFS) do not appear to take any advantage from the side information that u (resp. v) divides $p-1$ (resp. $q-1$). The same is true of simpler methods like Pollard's $p-1$ since we have ensured that neither $p-1$ nor $q-1$ is smooth. Finally, elliptic curve weaponry [18] will not pull-out factors of n in the range considered. Note that the requested size of n (768 bits or more) makes factoring n a very hard task anyway.

We now turn the size of σ. In order to avoid the computation of discrete logarithms by the baby step-giant step method, we have to make σ large enough. As already stated, 2^{160} is a minimum. This can be achieved for example by making σ a permutation of the first 30 odd primes, which yields $\sigma \simeq 2^{160.45}$. Alternatively, one one can choose a sequence of 16 primes with 10 bits. Since there are 75 such primes, this leads to a $\cong 53$-bit entropy. Adding prime powers, as stated above, will further increase these figures.

There is a further difficulty, when σ is known. Note that

$$4ab = \frac{\phi(n)}{\prod_{i=1}^{k} p_i} = \frac{n-p-q+1}{\sigma}$$

hence $4ab$ differs from $\frac{n}{\sigma}$ only by $\epsilon = -\frac{p+q-1}{\sigma}$. The numerator is of size $|n|/2$, hence, if it does not exceed the denominator by a fairly large number of bits, the value of ab is basically known and decryption can be performed.

When the exact splitting of the factors of σ into u and v are known as well, the previous analysis can be pushed further. Reducing the relation $n = (2au+1)(2bv+1)$ modulo u, we find that $n = 2bv + 1 \bmod u$ and we can calculate $d = b \bmod u$. Similarly, we learn $c = a \bmod v$. We let $a = rv + c$ and $b = su + d$, with r, s unknown and, using the fact that $\sigma = uv$, we obtain:

$$n = (2rvu + 2cu + 1)(2suv + 2dv + 1) =$$

$$4rs\sigma^2 + 2\sigma[r(2dv+1) + s(2cu+1)] + (2cu+1)(2dv+1)$$

which is of the form

$$n = 4rs\sigma^2 + 2\sigma(\alpha r + \beta s) + \gamma$$

with known α, β and γ. Reducing modulo σ^2, this provides the value δ of $\alpha r + \beta s \bmod \sigma$. At this point, our analysis becomes quite technical and the reader may skip the following and jump to the conclusion that $n >> \sigma^4$.

For the interested reader, we note that the pair (r, s) lies in the two-dimensional lattice L defined by

$$L = \{(x, y) | \alpha x + \beta y = \delta \bmod \sigma\}$$

This lattice has determinant σ. Also, it is easily seen that α and β are bounded by 2σ and γ by $4\sigma^2$. From this we get

$$rs \leq \frac{n}{4\sigma^2} \leq rs + r + s + 1 = (r+1)(s+1)$$

Thus, the pair (r, s) is very close to the boundary of the curve C with equation $xy = \frac{n}{4\sigma^2}$. More precisely, the distance between the pair (r, s) and the curve does not exceed $\sqrt{2}$. This defines a geometric area A that includes (r, s). Now, key generation usually induces constraints that limit the possible range of the parameters. For this reason, it is appropriate to replace C by the line $x + y = \frac{\sqrt{n}}{2\sigma}$ in order to estimate the size of A. This leads to an approximation which is $O(\frac{\sqrt{n}}{\sigma})$. The number of lattice points from L in this area is, in turn, measured by the ratio between the size of A and the determinant, which is $\frac{\sqrt{n}}{\sigma^2}$. It is safe to ensure that this set is beyond exhaustive search, which we express by $n >> \sigma^4$.

Note that the ratio $|n|/|\sigma|$ is the expansion rate of the encryption, where $|n|$ denotes, as usual, the size of n in bits. It is of course desirable to make this rate as low as possible. On the other hand, as a consequence of the above remarks, we see that $\frac{|n|}{4} - |\sigma|$ should be large. Asymptotically, this is achieved as soon as we fix an expansion rate which is > 4. For real-size parameters, we suggest to respect the heuristic bound $\frac{|n|}{4} - \sigma \geq 128$, which is consistent with our minimal parameters. Larger parameters allow a slightly better expansion rate.

2.5 Performances

Despite its expansion rate, the new cryptosystem is quite efficient: encryption requires the elevation of a constant 768-bit number to a 160-bit power. Several batch ([21, 23]) and pre-processing ([2]) techniques can speed-up such computations, which might be a small advantage over RSA.

Decryption is slightly more awkward since k exponentiations are needed. But this number can be reduced in a few ways :

Firstly, while computing $c^{\phi(n)/p_i} \bmod n$ for each i, it is possible to first store $c' = c^{4ab} \bmod n$ and raise c' to the successive powers σ/p_i so that (besides the first one), the remaining exponentiations involve 160-bit powers. One can further, in the square-and-multiply algorithm, share the "square" part of the various exponentiations. A careful bookkeeping of the number of modular multiplications obtained by setting $|n| = 768$ and choosing sixteen 10-bit primes p_i, shows that the total number of modular multiplications decreases to 2352: 912 for the computation of c' and 1440 for the rest Actually, the "multiply" part can be somehow amortized as well: we refer to [21] for a proper description of such an optimized exponentiation strategy. The resulting computing load is less than what is needed for a couple of RSA decryptions with a similar modulus.

Unfortunately, there is a drawback in reducing the value of k: in the 30-prime variant it is necessary to store 1718 different $t[i, j]$ hash values. Hashing on two bytes seems enough and results in an overall memory requirement of four kilobytes. In the 16-prime variant, hash values of 3 bytes are necessary and the table size becomes $\cong$ 100 kilobytes. As observed at the end of section 2.2, the hash table can be drastically reduced at the cost of a minute computation overhead.

Another speed-up can be obtained by separately performing decryption modulo p and q so as to take advantage

of smaller operand sizes. This alone, divides the decryption workfactor by four.

Finally, decryption is inherently parallel and naturally adapted to array processors since each m_i can be computed independently of all the others.

2.6 Implementation

The new scheme (768-bit n, $k = 30$) was actually implemented on a 68HC05-based ST16CF54 smart-card (4,096 EEPROM bytes, 16,384 ROM bytes and 352 RAM bytes). The public key is only 96-byte long and as in most smart-card implementations, n's storage is avoided by a command that re-computes the modulus from its factors upon request (re-computation and transmission take 10 ms). For further space optimization g's first 91 bytes are the byte-reversed binary complement of n's last 91 bytes. Decryption (a 4,119-byte routine) takes 3,912 ms. Benchmarks were done with a 5 MHz oscillator and ISO 7816-3 T=0 transmission at 115,200 bauds.

3 The probabilistic version

3.1 The setting

We now turn to the probabilistic version of the scheme. As already explained, we adopt a more complexity-oriented approach and, for example, we view B as bounded by a polynomial in $\log n$. The probabilistic version replaces the ciphertext $g^m \bmod n$ by $c = x^\sigma g^m \bmod n$, where x is chosen at random among positive integers $< n$. Decryption remains identical. This is due to the fact that the effect of multiplying by x^σ is cancelled by raising the ciphertext to the various powers $\frac{\phi(n)}{p_i}$, as performed by the decryption algorithm. Note that this version requires σ to be public.

The resulting scheme is *homomorphic*, which means that $E(m + m' \bmod \sigma) = E(m)E(m') \bmod n$. Probabilistic homomorphic encryption has received a lot of applications, both practically and theoretically oriented. To name a few, we quote the early work of Benaloh on election schemes ([4]) and the area of zero-knowledge proofs for NP (see [13, 3]). Known such schemes are the Quadratic Residuosity schemes of Goldwasser and Micali ([12]) which encrypts only one bit and its extensions to higher residues modulo a *single* prime (see [4]), which encrypts a few bits. As already explained in section 1, these schemes suffer from a serious drawback: a complexity theoretic analysis has to view the cleartext as logarithmic in the size of of ciphertext. In other words, the expansion rate, i.e. the ratio between the length of the ciphertext and the length of the cleartext is huge. In our proposal, this ratio is exactly $\frac{|n|}{|\sigma|}$. Note that that our assumption that σ is B-smooth, for some small B, does not preclude a linear ratio. The maximum size of σ is $\sum_{p<B} \log p$, where p ranges over primes and it is known that $\theta(B) = \sum_{p<B} \ln p \simeq B$. Thus, even if B is logarithmic in n, there are enough primes to make $|\sigma|$ a linear proportion of $|n|$. This is a definite improvement over previous homomorphic schemes. Note however that, following the comments in section 2.4, it is safe to take $\frac{|\sigma|}{|n|} < 1/4$.

3.2 A complexity theoretic approach

We already observed that the security of our proposal is related to the question of distinguishing higher residues modulo n, that is integers of the form $x^p \bmod n$, when p is a prime divisor of $\phi(n)$. In the rest of this section, we want to clarify this relationship in the asymptotic setting of complexity theory. In view of the remarks just made, we find it convenient to assume that the ratio $\frac{|\sigma|}{|n|}$ has a fixed value $\alpha < 1/4$. We also fix a polynomial B in $\log n$. The parameters which are of interest to us are pairs (n, σ) such that σ is squarefree, odd and B-smooth, n is a product of two primes p, q, σ is a divisor of $\phi(n)$ prime to $\phi(n)/\sigma$ and $\frac{|\sigma|}{|n|} = \alpha$. We call any integer n that appears as first coordinate of such a pair (B, α)-dense. Distinguishing higher residues is usually considered difficult (see [4]). We conjecture that this remains true when n varies over (B, α)-dense integers. Towards a more precise statement, let $R_p(y, n)$ be one if y is a p-th residue modulo n and zero otherwise. Define a higher residue oracle to be a probabilistic polynomial time algorithm A which takes as input a triple (n, y, p) and returns a bit $A(n, y, p)$ such that the following holds:
There exists a polynomial Q in $|n|$ such that, for infinitely many values of $|n|$, one can find a prime $p(|n|) < B$, with:

$$Pr\{A(n, y, p) = R_p(y, n)\} \geq 1 - \frac{1}{p} + \frac{1}{Q}$$

where the probability is taken over the random tosses of A and its inputs, conditionnally to the event that n is (B, α)-dense and p is a divisor of $\phi(n)$.

Our **Intractability Hypothesis** is that there is no higher residue oracle. The constant $1 - \frac{1}{p}$ comes from the obvious stategy for approximating R_p which consists in constantly outputting zero. This strategy is successful for a proportion $1 - \frac{1}{p}$ of the inputs.

3.3 A security proof

The security of probabilistic encryption scheme has been investigated in [12]. In this paper, the authors introduced the notion of *semantic security*: given two messages m_0 and m_1, a message distinguisher is a probabilistic polynomial time algorithm D, which distinguishes encryptions of m_0 from encryptions of m_1. More, accurately, it outputs a bit $D(n, \sigma, g, y)$ in such a way that, setting

$$\theta_i = Pr\{D(n, \sigma, g, y) = 1 | y \in E(m_i)\}$$

where $E(m_i)$ is the set of encryptions of m_i, the following holds:
There exists a polynomial Q in $|n|$ such that, for infinitely many values of $|n|$, $|\theta_0 - \theta_1| \geq \frac{1}{Q}$

Semantic security is the assertion that there is no pair of polynomial time algorithms F, D such that F produces two messages for which D is a message distinguisher.

Theorem 1 *Assume that no higher residue oracle exists. Then, the probabilitic version of the encryption scheme has semantic security.*

The proof of this result uses the *hybrid technique* for which we refer to [11]. It is technical in character and we have chosen to only include a sketch it in an appendix to the present paper.

4 Applications and variants

Even if we do not expect large scale replacement of RSA by our scheme, we feel that the latter is worth some academic interest. Especially, we believe that it opens up new applications. We have not yet fully investigated those potential applications but we give some suggestions below.

4.1 Traceability

Our proposal could offer some help in the management of key escrowing services. Consider the variant of the Diffie-Hellman key exchange protocol, where a composite modulus n is used. Such a variant has been studied by various researchers including Mc Curley in [20], where it is shown that some specific choices lead to a scheme that is at least as difficult as factoring. Assume further that the modulus n and the base for exponentiations g are chosen as described in section 1. It has been proposed (see e.g [14]) that g and n could be defined by some kind of TTP (Trusted Third Party). Now, the user's public key y and his secret key x are related by $y = g^x \bmod n$. It is conceivable to leave the choice of x to the user with the provision that $x \bmod \sigma = ID$, where ID is the identity of the user. This can be checked by the TTP upon registration of the key. Thus, we have reached a situation where the identity is embedded in the public key through a trapdoor, although the actual key is not. One should not however overestimate the resulting functionality. It could be useful in scenarios where traceability is made possible via escrowing but where confidentiality cannot be broken even with the help of the escrowing services. Alternatively, it might be used to split traceability and secret key recovery between key escrows. Note that the above proposal requests that σ is made public: as already observed, this does not seem to endanger the scheme.

4.2 Variants of the scheme

As is often the case, one can design numerous variants of the basic scheme. We will mention two because of their potential applications.

Use of moduli with three prime factors As for RSA, it is possible to embed three prime factors p, q, r in the modulus in place of two. The construction is straightforward: the small odd primes p_i are split into three groups thus yielding, by multiplication, three integers u, v, w. The three primes are then sought among integers of the form $2au + 1$ (resp. $2bv + 1$, resp. $2cw + 1$). It seems possible to keep the minimum size of n to 768 bits, which allows a, b, c to be around 200 bits. Following an idea of Maurer and Yacobi ([19]), we can then have a complete trapdoor for the discrete logarithm with base g: once the σ part has been computed, there remains to compute the logarithm modulo a, b and c, which is not immediate but well within the reach of current technology, since these numbers are 200 bit integers. Again, the variant could prove useful in key escrowing scenarios of, say, Diffie-Hellman keys, where it might be desirable to have a lengthy recovery of the secret key for consumer's protection.

Multiplicative encryption In this variant, σ is made public and encryption applies to messages of length k, $m = \sum_{i=1}^{k} m_i 2^{i-1}$. In order to encrypt m, one computes $e = \prod_{i=1}^{k} p_i^{m_i}$ and apply probabilistic encryption to e. Of course, the bandwith of this variant is very low: using a 768 bit modulus n and choosing the first 30 odd primes for p_is, we obtain a 30 bit input and a 768 bit output. Allowing a larger input has drastic consequences in terms of the size of n. The value of σ is close to 2^{560} when the first k primes are used with $k = 80$ but reaches $2^{993.4}$ for $k = 128$ and 2^{1309} for $k = 160$. Using the heuristic bound mentioned in section 2.4, we get for the length of n something beyond 5000 bits if k is 160. This goes down to 2400 bits when $k = 80$.

As a result, the variant just decribed is not really practical and there is little chance that it can ever be adopted as an actual encryption scheme. On the other hand, the ciphertext $c(m)$ can be used in an encryption scheme à la El Gamal. The modulus is not prime since it is an RSA modulus, but it makes no difference on the user's size. From $h = c(m)$, he can manufacture a public key y with a corresponding matching secret key x of his choice $y = h^x \bmod n$ The resulting cryptosystem allows ciphertext traceability in the sense of Desmedt (see [9]). Our proposal enables to trace ciphertexts by a technique similar to the one used by Desmedt, but decreases the size of the modulus from something like 10000 bits to 2500 bits. The tracing algorithm goes as follows: extract from an El Gamal encryption the part $u = h^r \bmod n$ and apply the decryption algorithm, treating u as a ciphertext. The decryption algorithm will basically find the original message m, which provides the identity of the user and from which h was built. Several errors may occur due to the fact that r might have some of the p_is as divisors: the corresponding decrypted values of m_i will be set to 1, regardless of their original values. The correct value can be found if a sample of ciphertexts are available or, alternatively, if an error-correction capacity has been added to m. Such an error-correction mechanism is highly advisable anyway in view of the attacks against software key escrow reported in [15].

Note that, one can further reduce the size of the exponent. This is because 40 bits may be considered enough for tracing purposes. The value of σ goes down to approximately 2^{233} and 1088 bits becomes an acceptable minimum length for the modulus.

5 Challenge

It is a tradition in the cryptographic community to offer cash rewards for successful cryptanalysis. More than a simple motivation means, such rewards also express the designers' confidence in their own schemes. As an incentive to the analysis of the new scheme, we therefore offer \$ $|n|$ to whoever will decrypt :

$c =$ 13370fe62d81fde356d1842fd7e5fc1ae5b9b449
bdd00866597e61af4fb0d939283b04d3bb73f91f
0d9d61eb0014690e567ab89aa8df4a9164cd4c6e
6df80806c7cdceda5cfda97bf7c42cc702512a49
dd196c8746c0e2ef36ca2aee21d4a36a$_{16}$

$g =$ 0b9cf6a789959ed4f36b701a5065154f7f4f1517
6d731b4897875d26a9e24415e111479050894ba7
c532ada1903c63a84ef7edc29c208a8ddd3fb5f7
d43727b730f20d8e12c17cd5cf9ab4358147cb62
a9fb8878bf15204e444ba6ade6132743$_{16}$

$n =$ 1459b9617b8a9df6bd54341307f1256dafa241bd
65b96ed14078e80dc6116001b83c5f88c7bbcb0b
db237daac2e76df5b415d089baa0fd078516e60e
2cdda7c26b858777604c5fbd19f0711bc75ce00a
5c37e2790b0d9d0ff9625c5ab9c7511d$_{16}$

where $k = 30$ (p_i is the i-th odd prime) and the message is ASCII-encoded. The challenger should be the first to decrypt at least 50% of c and publish the cryptanalysis method but the authors are ready to carefully evaluate *ad valorem* any feedback they get.

Acknowledgements

The paper grew out of a previous version which did not include the probabilistic case of our scheme. We wish to thank Julien Stern for suggesting us this alternative mode of encryption. We also want to thank J. Benaloh for help in clarifying our respective contributions in the definition of the probabilistic case. Finally, we are grateful to Adi Shamir, for helpful comments including the improved decryption algorithm mentioned in section 2.2 and also to one of the anonymous referees for pointing out the clever trick that yields the improved security analysis included at the end of section 2.4.

References

[1] R. Anderson, *Robustness principles for public-key protocols*, Advances in Cryptology Crypto'95, Santa Barbara, Lectures Notes in Computer Science 963, pp. 236–247, Springer-Verlag, 1995.

[2] E. Brickell, D. Gordon, K. McCurley and D. Wilson, *Fast Exponentiation with Precomputation*, Advances in Cryptology Eurocrypt'92, Balatonfüred, Lectures Notes in Computer Science 658, pp. 200–207, Springer-Verlag, 1993.

[3] G. Brassard, D. Chaum and C. Crépeau, *Minimum Disclosure Proofs of Knowledge*, JCSS, Vol. 37(2), Oct. 1988, pp. 156–189.

[4] J. D. Cohen Benaloh, *Verifiable Secret-Ballot Elections*, Ph-D thesis, Yale University, 1988.

[5] J. D. Cohen and M. J. Fischer, (1985), *A robust and verifiable cryptographically secure election scheme*, Proc. of 26th Symp. on Foundation of Computer Science, 1985, 372–382.

[6] J. D. Cohen Benaloh, *Cryptographic Capsules: A Disjunctive Primitive for Interactive Protocols*, Advances in Cryptology Crypto'86, Santa Barbara, Lectures Notes in Computer Science , pp. 213–222, Springer-Verlag, 1986.

[7] J. D. Cohen Benaloh and M. Yung, *Distributing the Power of a Government to Enhance the Privacy of Voters*, Proc. of 5h Symp. on Principles of Distributed Computing, 1986, 52–62.

[8] D. Denning (Robling), *Cryptography and data security*, Addison-Wesley Publishing Company, pp. 148, 1983.

[9] Y. Desmedt, *Securing traceability of ciphertexts – Towards a secure software key escrow system*, Advances in Cryptology Eurocrypt'95, Saint-Malo, Lectures Notes in Computer Science 921, pp. 417–157, Springer-Verlag, 1995.

[10] W. Diffie and M. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory, vol. IT-22-6, pp. 644–654, 1976.

[11] O. Goldreich, *Foundations of cryptography (Fragments of a book)*. Weizmann Institut of Science, 1995.

[12] S. Goldwasser and S. Micali, *Probabilistic Encryption*, JCSS, 28(2), April 1984, pp. 270–299.

[13] O. Goldreich, S. Micali and A. Wigderson, *Proofs that Yield Nothing but their Validity and a Methodology of Cryptographic Protocol Design*, Proc. of 27th Symp. on Foundation of Computer Science, 1986, pp.174–187.

[14] N. Jefferies, C. Mitchell and M. Walker, *A proposed architecture for trusted third party services*, Cryptography Policy and Algorithms, Queensland, Lecture Notes in Computer Science 1029, pp. 98–114, Springer-Verlag, 1996.

[15] L. Knudsen and T. Pedersen, *On the difficulty of software key escrow*, Advances in Cryptology Eurocrypt'96, Saragossa, Lectures Notes in Computer Science 1070, pp. 237–244, Springer-Verlag, 1996.

[16] P. Kocher, *Timing attacks in implementations of Diffie-Hellman, RSA, DSS and other systems*, Advances in Cryptology Crypto'96, Santa Barbara, Lectures Notes in Computer Science , pp. 104-113, Springer-Verlag, 1996.

[17] Kaoru Kurosawa, Yutaka Katayama, Wakaha Ogata and Shigeo Tsujii, *General public key residue cryptosystems and mental poker protocols*, Advances in Cryptology Eurocrypt'90, Aarhus, Lectures Notes in Computer Science 473, pp. 374–388, Springer-Verlag, 1996.

[18] H. Lenstra Jr., *Factoring integers with elliptic curves*, Annals of Mathematics, 126, pp. 649–673, 1991.

[19] U. Maurer and Y. Yacobi, *Non-interactive public key cryptography*, Advances in Cryptology Eurocrypt'91, Brighton, Lectures Notes in Computer Science 547, pp. 498–507, Springer-Verlag, 1991.

[20] K. McCurley, *A key distribution system equivalent to factoring*, Journal of Cryptology, vol. 1, pp. 85–105, 1988.

[21] D. M'Raïhi and D. Naccache, *Batch exponentiation - A fast DLP-based signature generation strategy*, Proceedings of the third ACM conference on Computer and Communications Security, New Delhi, pp. 58–61 , 1996.

[22] T. Okamoto and S. Uchiyama, *A new public-key cryptosystem as secure as factoring*, Advances in Cryptology Eurocrypt'98, Helsinki, Lectures Notes in Computer Science , pp. to appear, Springer-Verlag, 1998.

[23] D. Naccache and J. Stern, *A new public-key cryptosystem*, Advances in Cryptology Eurocrypt'97, Constance, Lectures Notes in Computer Science 1233, pp. 27–36, Springer-Verlag, 1997.

[24] Sung-Jun Park and Dong-Ho Won, *A generalization of public key residue cryptosystem*, In Proc. of 1993 KOREA-JAPAN joint workshop on information security and cryptology, 202–206.

[25] B. Pfitzmann and M. Schunter, *Asymmetric fingerprinting*, Advances in Cryptology Eurocrypt'96, Saragossa, Lectures Notes in Computer Science 1070, pp. 84–95, Springer-Verlag, 1996.

[26] D. Pointcheval, *A new identification scheme based on the perceptrons problem*, Advances in Cryptology Eurocrypt'94, Perugia, Lectures Notes in Computer Science 950, pp. 318–328, Springer-Verlag, 1995.

[27] S. C. Pohlig and M. E. Hellman, *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance* IEEE Transactions on Information Theory, vol. IT-24-1, pp. 106–110, 1978.

[28] J. Pollard, *Theorems on factorization and primality testing*, Proceedings of the Cambridge Philosophical Society, vol. 76, pp. 521-528, 1974.

[29] J. Pollard, *Factoring with cubic integers*, A. Lenstra and H. Lenstra Jr., The development of the number field sieve, vol. 1554, LNM, 4–10, Springer-Verlag, 1993.

[30] C. Pomerance, *Analysis and comparison of some integer factoring algorithms*, printed in H. Lenstra Jr. and R. Tijdeman, Computational Methods in Number Theory I, Mathematisch Centum Tract 154, Amsterdam, pp. 89–139, 1982.

[31] M. Rabin, *Digitalized signatures and public-key functions as intractable as factorization*, MIT/LCS/TR-212, MIT Laboratory for Computer Science, 1979.

[32] R. Rivest, A. Shamir and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, vol. 21-2, pp. 120-126, 1978.

[33] A. Shamir, *An efficient identification scheme based on permuted kernels*, Advances in Cryptology Crypto'89, Santa Barbara, Lectures Notes in Computer Science 435, pp. 606–609, Springer-Verlag, 1990.

[34] A. Shamir, *RSA for paranoids*, CryptoBytes, vol. 1-3, pp. 1–4, 1995.

[35] J. Stern, *A new identification scheme based on syndrome decoding*, Advances in Cryptology Crypto'93, Santa Barbara, Lectures Notes in Computer Science 773, pp. 13–21, Springer-Verlag, 1994 .

[36] J. Stern, *Designing identification schemes with keys of short size*, Advances in Cryptology Crypto'94, Santa Barbara, Lectures Notes in Computer Science 839, pp. 164–173, Springer-Verlag, 1995.

Appendix: Sketch of the Security Proof.

We show that any message distinguisher can be turned into an algorithm that recognizes higher residues. We let D be a distinguisher for two messages m_0 and m_1 and start from the fact that, keeping the above notations, θ_0 and θ_1 are significantly distinct. We next use the *hybrid technique* for which we refer to [11], pp.91–93. Hybrids consist of a sequence of random variables Y_i, $0 \leq i \leq k$, such that

1. Extreme hybrids collide with $E(m_0)$ and $E(m_1)$ respectively.
2. Random values of each hybrid can be produced by a probabilistic polynomial time algorithm.
3. There are only polynomially many hybrids.

In such a situation, [11] shows that D distinguishes two neighbouring hybrids. Our hybrids are formed by considering a message μ_i, such that

$$\mu_i = m_0 \bmod p_j \quad \text{for} \quad j > i \text{ and}$$

$$\mu_i = m_1 \bmod p_j \quad \text{for} \quad j \leq i$$

and letting Y_i to be uniformly distributed over the set $E(\mu_i)$ of encryptions of μ_i. It is easily seen that conditions 1, 2 and 3 are satisfied. Thus, for some index i, D significantly distinguishes Y_i and Y_{i-1}. Set $\mu = \mu_i$, $p = p_i$ and let μ^j, $1 \leq j \leq p$, be the unique message such that

$$\mu^j = \mu \bmod p_\ell \quad \text{for} \quad \ell \neq i \text{ and } \mu^j = j \bmod p$$

We note that, both m_i and m_{i-1} appear among the μ^js and we show that D cannot distinguish encryptions of any two of the μ^js. This will yield the desired contradiction.

Let

$$\pi_j = Pr\{D(n, \sigma, g, y) = 1 | y \in E(\mu_j)\}$$

and assume that some π_i significantly exceeds the other ones. In other words, $\pi_i \geq \sup_{j \neq i} \pi_j + \frac{1}{Q}$ for some polynomial Q and infinitely many values of $|n|$. We show how to predict p-th residuosity: given z, we run D over a large sample N of inputs (n, σ, y) where $y = x^\sigma z^{t\sigma/p} g^{\mu_i}$, with $x > n$ and $\ell \leq p$ chosen at random, and we average the outputs. Now, if z is a p-th residue, then y simply varies over $E(\mu_i)$, whereas, if z is not a p-th residue, y randomly varies over the union of all $E(\mu_j)$s. Thus, in the first case, the average is close to π_i, whereas, in the second case, it is approximately $\frac{\sum_{j=1}^{p} \pi_j}{p}$. It is easily seen that the difference is bounded from below by $\frac{p-1}{p}\frac{1}{Q}$. Using the law of large numbers, this is enough to make the proper decision on the p-th residuosity, with probability as close to 1 as we wish, by using only polynomially large samples. This finishes the proof.

Remarks.

1. Turning the previous sketch into a complete proof involves a technical but rather long write-up: especially, a precise version of the law of large numbers has to be made explicit, e.g. by using the Chebishev inequality. Also, the values of π_i and $\frac{\sum_{j=1}^{p} \pi_j}{p}$ are not known *a priori* and should be approximated as well using the law of large numbers. We urge the interested reader to consult [11] for similar proofs.
2. The higher residuosity oracle that was built in the proof for the sake of contradiction uses inputs σ and g on top of n, y and p. Actually, one can check that everything goes through, *mutatis mutandis*, if σ is replaced by $\overline{\sigma} = \prod_{p<B} p$. Thus σ is not really needed. As for g, as seen in section 2.1, it can be chosen at random: a proper choice will be spotted by sampling the corresponding oracle and checking its correctness.

Twin Signatures: an Alternative to the Hash-and-Sign Paradigm

David Naccache
Gemplus Card International
34, rue Guynemer
92447 Issy-les-Moulineaux, France
david.naccache@gemplus.com

David Pointcheval Jacques Stern
École Normale Supérieure
45, rue d'Ulm
75230 Paris cedex 05, France
{david.pointcheval,jacques.stern}@ens.fr

ABSTRACT

This paper introduces a simple alternative to the hash-and-sign paradigm, from the security point of view but for signing short messages, called *twinning*. A twin signature is obtained by signing twice a short message by a signature scheme. Analysis of the concept in different settings yields the following results:

- We prove that no generic algorithm can efficiently forge a twin DSA signature. Although generic algorithms offer a less stringent form of security than computational reductions in the standard model, such successful proofs still produce positive evidence in favor of the correctness of the new paradigm.
- We prove in standard model an equivalence between the hardness of producing existential forgeries (even under adaptively chosen message attacks) of a twin version of a signature scheme proposed by Gennaro, Halevi and Rabin and the Flexible RSA Problem.

We consequently regard twinning as an interesting alternative to hash functions for eradicating existential forgery in signature schemes.

Keywords

Digital Signatures, Provable Security, Discrete Logarithm, Generic Model, Flexible RSA Problem, Standard Model.

1. INTRODUCTION

The well-known *hash and sign* paradigm has two distinct goals: increasing *performance* by reducing the size of the signed message and improving *security* by preventing existential forgeries. As a corollary, hashing remains mandatory even for short messages.

From the conceptual standpoint, the use of hash functions comes at the cost of extra assumptions such as the conjecture that for all practical purposes, concrete functions can be identified with ideal black boxes [3] or that under certain circumstances (black box groups [15, 21]) a new group element must necessarily come from the addition of two *already known* elements. In some settings [11] both models are even used simultaneously.

This paper investigates a simple substitute to hashing that we call *twinning*. A twin signature is obtained by signing twice the same (short) raw message by a probabilistic signature scheme, or two probabilistically related messages.

We believe that this simple paradigm is powerful enough to eradicate existential forgery in a variety of contexts. To support this claim, we show that no generic algorithm can efficiently forge a twin DSA signature and prove that for a twin variant of a signature scheme proposed by Gennaro, Halevi and Rabin [8] (hereafter GHR) existential forgery, even under an adaptively chosen-message attack, is equivalent to the Flexible RSA Problem [5] in the standard model.

CCS'01, November 5-8, 2001, Philadelphia, Pennsylvania, USA.

2. DIGITAL SIGNATURE SCHEMES

Let us begin with a quick review of definitions and security notions for digital signatures. Digital signature schemes are the electronic version of handwritten signatures for digital documents: a user's signature on a message m is a string which depends on m, on public and secret data specific to the user and–possibly–on randomly chosen data, in such a way that anyone can check the validity of the signature by using public data only. The user's public data are called the *public key*, whereas his secret data are called the *secret key*. The intuitive security notion would be the impossibility to forge user's signatures without the knowledge of his secret key. In this section, we give a more precise definition of signature schemes and of the possible attacks against them (most of those definitions are based on [9]).

2.1 Definitions

A signature scheme is defined by the three following algorithms:

- The *key generation algorithm* G. On input 1^k, where k is the security parameter, the algorithm G produces a pair (k_p, k_s) of matching public and secret keys. Algorithm G is probabilistic.
- The *signing algorithm* Σ. Given a message m and a pair of matching public and secret keys (k_p, k_s), Σ produces a signature σ. The signing algorithm might be probabilistic.

- The *verification algorithm* V. Given a signature σ, a message m and a public key $\mathsf{k_p}$, V tests whether σ is a valid signature of m with respect to $\mathsf{k_p}$. In general, the verification algorithm need not be probabilistic.

2.2 Forgeries and Attacks

In this subsection, we formalize some security notions which capture the main practical situations. On the one hand, the **goals** of the adversary may be various:

- Disclosing the secret key of the signer. It is the most serious attack. This attack is termed *total break.*
- Constructing an efficient algorithm which is able to sign messages with good probability of success. This is called *universal forgery.*
- Providing a new message-signature pair. This is called *existential forgery.*

In many cases this latter forgery, the *existential forgery*, is not dangerous, because the output message is likely to be meaningless. Nevertheless, a signature scheme which is not existentially unforgeable (and thus that admits existential forgeries) does not guarantee by itself the identity of the signer. For example, it cannot be used to certify randomly looking elements, such as keys. Furthermore, it cannot formally guarantee the non-repudiation property, since anyone may be able to produce a message with a valid signature.

On the other hand, various **means** can be made available to the adversary, helping her into her forgery. We focus on two specific kinds of attacks against signature schemes: the *no-message attacks* and the *known-message attacks.* In the first scenario, the attacker only knows the public key of the signer. In the second one, the attacker has access to a list of valid message-signature pairs. According to the way this list was created, we usually distinguish many subclasses, but the strongest is the *adaptively chosen-message attack*, where the attacker can ask the signer to sign any message of her choice. She can therefore adapt her queries according to previous answers.

When one designs a signature scheme, one wants to computationally rule out existential forgeries even under adaptively chosen-message attacks, which is the strongest security level for a signature scheme.

3. GENERIC ALGORITHMS

Before we proceed, let us stress that although the generic model in which we analyze DSA offers a somehow weaker form of security than the reductions that we apply to GHR in the standard model, it still provides *evidence* that twinning may indeed have a beneficial effect on security.

Generic algorithms [15, 21], as introduced by Nechaev and Shoup, encompass group algorithms that do not exploit any special property of the encodings of group elements other than the property that each group element is encoded by a unique string. Typically, algorithms like Pollard's ρ algorithm [18] fall under the scope of this formalism while index-calculus methods do not.

3.1 The Framework

Recall that any Abelian finite group Γ is isomorphic to a product of cyclic groups of the form $(\mathbb{Z}_{p^k}, +)$, where p is a prime. Such groups will be called standard Abelian groups. An encoding of a standard group Γ is an injective map from Γ into a set of bit-strings S.

We give some examples: consider the multiplicative group of invertible elements modulo some prime q. This group is cyclic and isomorphic to the standard additive group $\Gamma = \mathbb{Z}_{q-1}$. Given a generator g, an encoding σ is obtained by computing the binary representation $\sigma(x)$ of $g^x \bmod q$. The same construction applies when one considers a multiplicative subgroup of prime order r. Similarly, let E be the group of points of some non-singular elliptic curve over a finite field $\mathbb{F}$, then E is either isomorphic to a (standard) cyclic group Γ or else is isomorphic to a product of two cyclic groups $\mathbb{Z}_{d_1} \times \mathbb{Z}_{d_2}$. In the first case, given a generator G of E, an encoding is obtained by computing $\sigma(x) = x.G$, where $x.G$ denotes the scalar multiplication of G by the integer x and providing coordinates for $\sigma(x)$. The same construction applies when E is replaced by one of its subgroups of prime order r. Note that the encoding set appears much larger than the group size, but compact encodings using only one coordinate and a sign bit ± 1 exist and for such encodings, the image of σ is included in the binary expansions of integers $< tr$ for some small integer t, provided that r is close enough to the size of the underlying field $\mathbb{F}$. This is exactly what is recommended for cryptographic applications [10].

A *generic* algorithm $\mathcal{A}$ over a standard Abelian group Γ is a probabilistic algorithm that takes as input an *encoding list* $\{\sigma(x_1), \cdots, \sigma(x_k)\}$, where each x_i is in Γ. While it executes, the algorithm may consult an oracle for further encodings. Oracle calls consist of triples $\{i, j, \epsilon\}$, where i and j are indices of the encoding list and ϵ is $\pm$. The oracle returns the string $\sigma(x_i \pm x_j)$, according to the value of ϵ and this bit-string is appended to the list, unless it was already present. In other words, $\mathcal{A}$ cannot access an element of Γ directly but only through its name $\sigma(x)$ and the oracle provides names for the sum or difference of two elements addressed by their respective names. Note however that $\mathcal{A}$ may access the list at any time. In many cases, $\mathcal{A}$ takes as input a pair $\{\sigma(1), \sigma(x)\}$. Probabilities related to such algorithms are computed with respect to the internal coin tosses of $\mathcal{A}$ as well as the random choices of σ and x.

The following theorem appears in [21]:

THEOREM 1. *Let Γ be a standard cyclic group of order N and let p be the largest prime divisor of N. Let $\mathcal{A}$ be a generic algorithm over Γ that makes at most n queries to the oracle. If $x \in \Gamma$ and an encoding σ are chosen at random, then the probability that $\mathcal{A}$ returns x on input $\{\sigma(1), \sigma(x)\}$ is $\mathcal{O}(n^2/p)$.*

PROOF. We refer to [21] for a proof. However, we will need, as an ingredient for our own proofs, the probabilistic model used by Shoup. We develop the model in the special case where N is a prime number r, which is of interest to us. Alternatively, we could work in a subgroup of prime order r.

Basically, we would like to identify the probabilistic space consisting of σ and x with the space $S^{n+2} \times \Gamma$, where S is the set of bit-string encodings. Given a tuple $\{z_1, \cdots, z_{n+2}, y\}$ in this space, z_1 and z_2 are used as $\sigma(1)$ and $\sigma(x)$, the successive z_i are used in sequence to answer the oracle queries and the unique value y from Γ serves as x. However, this interpretation may yield inconsistencies as it does not take care of possible collisions between oracle queries. To overcome the difficulty, Shoup defines, along with the execution

of $\mathcal{A}$, a sequence of linear polynomials $F_i(X)$, with coefficients modulo r. Polynomials F_1 and F_2 are respectively set to $F_1 = 1$ and $F_2 = X$ and the definition of polynomial F_ℓ is related to the ℓ-th query $\{i, j, \epsilon\}$: $F_\ell = F_i \pm F_j$, where the sign $\pm$ is chosen according to ϵ. If F_ℓ is already listed as a previous polynomial F_h, then F_ℓ is marked and $\mathcal{A}$ is fed with the answer of the oracle at the h-th query. Otherwise, z_ℓ is returned by the oracle. Once $\mathcal{A}$ has come to a stop, the value of x is set to y.

It is easy to check that the behavior of the algorithm which plays with the polynomials F_i is exactly similar to the behavior of the regular algorithm, if we require that y is not a root of any polynomial $F_i - F_j$, where i, j range over indices of unmarked polynomials. A sequence $\{z_1, \cdots, z_{n+2}, y\}$ for which this requirement is met is called a *safe* sequence. Shoup shows that, for any $\{z_1, \cdots, z_{n+2}\}$, the set of y such that $\{z_1, \cdots, z_{n+2}, y\}$ is not safe has probability $\mathcal{O}(n^2/r)$. From a safe sequence, one can define x as y and σ as any encoding which satisfies $\sigma(F_i(y)) = z_i$, for all unmarked F_i. This correspondence preserves probabilities. However, it does not completely cover the sample space $\{\sigma, x\}$ since executions such that $F_i(x) = F_j(x)$, for some indices i, j, such that F_i and F_j are not identical are omitted. To conclude the proof of the above theorem in the special case where N is a prime number r, we simply note that the output of a computation corresponding to a safe sequence $\{z_1, \cdots, z_{n+2}, y\}$ does not depend on y. Hence it is equal to y with only minute probability. □

3.2 Digital Signatures over Generic Groups

We now explain how generic algorithms can deal with attacks against DSA-like signature schemes [6, 20, 16, 10]. We do this by defining a generic version of DSA that we call GDSA. Parameters for the signature include a standard cyclic group of prime order r together with an encoding σ. The signer also uses as a secret key/public key pair $\{x, \sigma(x)\}$. Note that we have chosen to describe signature generation as a regular rather than generic algorithm, using a full description of σ. To sign a message m, $1 < m < r$ the algorithm executes the following steps:

1. Generate a random number u, $1 \leq u < r$.
2. Compute $c \leftarrow \sigma(u) \bmod r$. If $c = 0$ go to step 1.
3. Compute $d \leftarrow u^{-1}(m + xc) \bmod r$. If $d = 0$ go to step 1.
4. Output the pair $\{c, d\}$ as the signature of m.

The verifier, on the other hand, is generic:

1. If $c \notin [1, r-1]$ or $d \notin [1, r-1]$, output invalid and stop.
2. Compute $h \leftarrow d^{-1} \bmod r$, $h_1 \leftarrow hm \bmod r$ and $h_2 \leftarrow hc \bmod r$.
3. Obtain $\sigma(h_1 + h_2 x)$ from the oracle and compute $c' \leftarrow \sigma(h_1 + h_2 x) \bmod r$.
4. If $c \neq c'$ output invalid and stop otherwise output valid and stop.

The reader may wonder how the verifier obtains the value of σ requested at step 3. This is simply achieved by mimicking the usual double-and-add algorithm and asking the appropriate queries to the oracle. This yields $\sigma(h_1)$ and $\sigma(h_2 x)$. A final call to the oracle completes the task.

A generic algorithm $\mathcal{A}$ can also perform forgery attacks against a signature scheme. This is defined by the ability of $\mathcal{A}$ to return on input $\{\sigma(1), \sigma(x)\}$ a triple $\{m, c, d\} \in \Gamma^3$ for which the verifier outputs valid. Here we assume that both algorithms are performed at a stretch, keeping the same encoding list.

To deal with adaptive attacks one endows $\mathcal{A}$ with another oracle, called the signing oracle. To query this oracle, the algorithm provides an element $m \in \Gamma$. The signing oracle returns a valid signature $\{c, d\}$ of m. Success of $\mathcal{A}$ is defined by its ability to produce a valid triple $\{\tilde{m}, \tilde{c}, \tilde{d}\}$, such that $\tilde{m}$ has not been queried during the attack.

Such a forgery can be easily performed against this GDSA scheme, even with just a passive attack: the adversary chooses random numbers h_1 and h_2, $1 \leq h_1, h_2 < r$ and computes $c \leftarrow \sigma(h_1 + h_2 x) \bmod r$. Then it defines $d = ch_2^{-1} \bmod r$, $h = d^{-1} \bmod r$, and eventually $m = dh_1 \bmod r$. The triple $\{m, c, d\} \in \Gamma^3$ is therefore a valid one, unless $c = 0$, which is very unlikely.

4. THE SECURITY OF TWIN GDSA

4.1 A Theoretical Result

The above definitions extend to the case of twin signatures, by requesting the attacker $\mathcal{A}$ to output an m and two distinct pairs $\{c, d\} \in \Gamma^2$, $\{c', d'\} \in \Gamma^2$. Success is granted as soon as the verifying algorithm outputs valid for both triples[1]. We prove the following:

THEOREM 2. *Let Γ be a standard cyclic group of prime order r. Let S be a set of bit-string encodings of cardinality at least r, included in the set of binary representations of integers $< tr$, for some t. Let $\mathcal{A}$ be a generic algorithm over Γ that makes at most n queries to the oracle. If $x \in \Gamma$ and an encoding σ are chosen at random, then the probability that $\mathcal{A}$ returns a message m together with two distinct GDSA signatures of m on input $\{\sigma(1), \sigma(x)\}$ is $\mathcal{O}(tn^2/r)$.*

PROOF. We cover the non adaptive case and tackle the more general case after the proof. We use the probabilistic model developed in section 3.1. Let $\mathcal{A}$ be a generic attacker able to forge some m and two distinct signatures $\{c, d\}$ and $\{c', d'\}$. We assume that, once these outputs have been produced, $\mathcal{A}$ goes on checking both signatures; we estimate the probability that both are valid.

We restrict our attention to behaviors of the full algorithm corresponding to safe sequences $\{z_1, \cdots, z_{n+2}, y\}$. By this, we discard a set of executions of probability $\mathcal{O}(n^2/r)$. We let P be the polynomial $(md^{-1}) + (cd^{-1})X$ and Q be the polynomial $(md'^{-1}) + (c'd'^{-1})X$.

- We first consider the case where either P or Q does not appear in the F_i list before the signatures are produced. If this happens for P, then P is included in the F_i list at signature verification and the corresponding answer of the oracle is a random number z_i. Unless $z_i = c \bmod r$, which is true with probability at most

[1] using [14] the simultaneous square-and-multiply generation or verification of two DSA signatures is only 17% slower than the generation or verification of a single signature.

t/r, the signature is invalid. A similar bound holds for Q.

- We now assume that both P and Q appear in the F_i list before $\mathcal{A}$ outputs its signatures. We let i denote the first index such that $F_i = P$ and j the first index such that $F_j = Q$. Note that both F_i and F_j are unmarked (as defined in section 3.1). If $i = j$, then we obtain that $md^{-1} = md'^{-1}$ and $cd^{-1} = c'd'^{-1}$. From this, it follows that $c = c'$, $d = d'$ and the signatures are not distinct.

- We are left with the case where $i \neq j$. We let $\Omega_{i,j}$, $i < j$, be the set of safe sequences producing two signatures such that the polynomials P, Q, defined as above appear for the first time before the algorithm outputs the signatures, as F_i and F_j. We consider a fixed value w for $\{z_1, \cdots, z_{j-1}\}$ and let $\hat{w}$ be the set of safe sequences extending w. We note that F_i and F_j are defined from w and we write $F_i = a + bX$, $F_j = a' + b'X$. We claim that $\Omega_{i,j} \cap \hat{w}$ has probability $\leq t/r$. To show this, observe that one of the signatures that the algorithm outputs is necessarily of the form $\{c, d\}$, with $c = z_i \bmod r$, $c = db \bmod r$ and $m = da \bmod r$. Now, the other signature is $\{c', d'\}$ and since m is already defined we get $d' = ma'^{-1} \bmod r$ and $c' = b'd' \bmod r$. This in turn defines $z_j \bmod r$ within a subset of at most t elements. From this, the required bound follows and, from the bound, we infer that the probability of $\Omega_{i,j}$ is at most t/r.

Summing up, we have bounded the probability that a safe sequence produces an execution of $\mathcal{A}$ outputting two valid signatures by $\mathcal{O}(tn^2/r)$. This finishes the proof. □

In the proof, we considered the case of an attacker forging a message-signature pair from scratch. A more elaborate scenario corresponds to an attacker who can adaptively request twin signatures corresponding to messages of his choice. In other words, the attacker interacts with the legitimate signer by submitting messages selected by its program.

We show how to modify the security proof that was just given to cover the adaptive case. We assume that each time it requests a signature the attacker $\mathcal{A}$ immediately verifies the received signature. We also assume that the verification algorithm is normalized in such a way that, when verifying a signature $\{c, d\}$ of a message m, it asks for $\sigma((md^{-1}) + (cd^{-1})x)$ after a fixed number of queries, say q. We now explain how to simulate signature generation: as before, we restrict our attention to behaviors of the algorithm corresponding to safe sequences $\{z_1, \cdots, z_{n+2}, y\}$. When the (twin) signature of m is requested at a time of the computation when the encoding list contains i elements, one picks z_{i+q} and z_{i+2q} and manufactures the two signatures as follows:

1. Let $c \leftarrow z_{i+q} \bmod r$, pick d at random.
2. Let $c' \leftarrow z_{i+2q} \bmod r$, pick d' at random.
3. Output $\{c, d\}$ and $\{c', d'\}$ as the first and second signatures.

While verifying both signatures, $\mathcal{A}$ will receive the elements z_{i+q} and z_{i+2q}, as

$$\sigma((md^{-1}) + (cd^{-1})x) \text{ and } \sigma((md'^{-1}) + (c'd'^{-1})x)$$

respectively, unless F_{i+q} or F_{i+2q} appears earlier in the F_i list. Due to the randomness of d and d', this happens with very small probability bounded by n/r. Altogether, the simulation is spotted with probability $\mathcal{O}(n^2/r)$ which does not affect the $\mathcal{O}(tn^2/r)$ bound for the probability of successful forgery.

4.2 Practical Meaning of the Result

We have shown that, in the setting of generic algorithms, existential forgery against twin GDSA has a minute success probability. Of course this does not tell anything on the security of actual twin DSA. Still, we believe that our proof has some *practical* meaning. The analogy with hash functions and the random oracle model [3] is inspiring: researchers and practitioners are aware that proofs in the random oracle model are not proofs but a mean to spot design flaws and validate schemes that are supported by such proofs. Still, all standard signature schemes that have been proposed use specific functions which are not random by definition; our proofs seem to indicate that if existential forgery against twin DSA is possible, it will require to dig into structural properties of the encoding function. This is of some help for the design of actual schemes: for example, the twin DSA described in Appendix A allows signature with message recovery without hashing and without any form of redundancy, while keeping some form of provable security. This might be considered a more attractive approach than [17] or [1], the former being based on redundancy and the latter on random oracles. We believe that twin DSA is even more convincing in the setting of elliptic curves, where there are no known ways of taking any advantage of the encoding function.

5. AN RSA-BASED TWINNING IN THE STANDARD MODEL

The twin signature scheme described in this section belongs to the (very) short list of efficient schemes provably secure in the standard model: in the sequel, we show that producing existential forgeries even under an adaptively chosen-message attack is equivalent to solving the Flexible RSA Problem [5].

Security in the standard model implies no ideal assumptions; in other words we directly reduce the Flexible RSA Problem to a forgery. As a corollary, we present an efficient and provably secure signature scheme that does not require any hash function.

Furthermore, the symmetry provided by twinning is much simpler to analyze than Cramer-Shoup's proposal [5] which achieves a similar security level, and similar efficiency, with a rather intricate proof.

5.1 Gennaro-Halevi-Rabin Signatures

In [8] Gennaro, Halevi and Rabin present the following signature scheme: Let n be an ℓ-bit RSA modulus [19], H a hash-function and $y \in \mathbb{Z}_n^*$. The pair $\{n, y\}$ is the signer's public key, whose secret key is the factorization of n.

- To sign m, the signer hashes $e \leftarrow H(m)$ (which is very likely to be co-prime with $\varphi(n)$) and computes the e-th

root of y modulo n using the factorization of n:

$$s \leftarrow y^{1/e} \bmod n$$

- To verify a given $\{m, s\}$, the verifier checks that

$$s^{H(m)} \bmod n \stackrel{?}{=} y.$$

Security relies on the Strong RSA Assumption. Indeed, if H outputs elements that contain at least a new prime factor, existential forgery is impossible. Accordingly, Gennaro *et al.* define a new property that H must satisfy to yield secure signatures: *division intractability*. Division intractability means that it is computationally impossible to find $a_1, \ldots, a_k$ and b such that $H(b)$ divides the product of all the $H(a_i)$. In [8], it is conjectured that such functions exist and heuristic conversions from collision-resistant into division-intractable functions are shown (see also [4]).

Still, security against adaptively chosen-message attacks requires the hash function H to either behave like a random oracle model or achieve the chameleon property [12]. This latter property, for a hash function, provides a trapdoor which helps to find second preimages, even with some fixed part. Indeed, some signatures can be pre-computed, but with specific exponents before outputting y: $y = x^{\prod_i e_i} \bmod n$ for random primes $e_i = H(m_i, r_i)$.

Using the chameleon property, for the i-th query m to the signing oracle, the simulator who knows the trapdoor can get an r such that $H(m_i, r_i) = H(m, r) = e_i$. In the random oracle model, one simply defines $H(m, r) \leftarrow e_i$.

Then $s = x^{\prod_{j \neq i} e_j} = y^{1/e_i} \bmod n$ and the signature therefore consists of the triple $\{m, r, s\}$ satisfying

$$s^{H(m,r)} = y \bmod n.$$

Cramer and Shoup [5] also proposed a scheme based on the Strong RSA Assumption, the first practical signature scheme to be secure in the standard model, but with universal one-way hash functions; our twin scheme will be similar but with a nice symmetry in the description (which helps for the security analysis) and no hash-functions, unless one wants to sign a long message.

5.2 Preliminaries

We build our scheme in two steps. The first scheme resists existential forgeries when subjected to no-message attacks. Twinning will immune it against adaptively chosen-message attacks.

5.2.1 *Injective function into the prime integers.*

Before any description, we will assume the existence of a function p with the following properties: given a security parameter k (which will be the size of the signed messages), p maps any string from $\{0,1\}^k$ into the set of the prime integers, p is also designed to be easy to compute and injective. A candidate is proposed and analyzed in Appendix B.

5.2.2 *The Flexible RSA Problem and the Strong RSA Assumption.*

Let us also recall the *Flexible RSA Problem* [5]. Given an RSA modulus n and an element $y \in \mathbb{Z}_n^\star$, find any exponent $e > 1$, together with an element x such that $x^e = y \bmod n$.

The *Strong RSA Assumption* is the conjecture that this problem is intractable for large moduli. This was independently introduced by [2, 7], and then used in many further security analyses (*e.g.* [5, 8]).

5.3 A First GHR Variant

The first scheme is very similar to GHR without random oracles but with function p instead:

- To sign $m \in \{0,1\}^k$, the signer computes $e \leftarrow p(m)$ and the e-th root of y modulo n using the factorization of n

$$s \leftarrow y^{1/e} \bmod n$$

- To verify a given $\{m, s\}$, the verifier checks that

$$s^{p(m)} \bmod n \stackrel{?}{=} y.$$

Since p provides a new prime for each new message (injectivity), existential forgery contradicts the Strong RSA Assumption. However, how can we deal with adaptively chosen-message attacks without any control over the output of the function p, which is a publicly defined non-random oracle and not a trapdoor function either?

5.4 The Twin Version

The final scheme is quite simple since it consists in duplicating the previous one: the signer uses two ℓ-bit RSA moduli n_1, n_2 and two elements y_1, y_2 in $\mathbb{Z}_{n_1}^\star$ and $\mathbb{Z}_{n_2}^\star$ respectively. Secret keys are the prime factors of the n_i.

- To sign a message m, the signer probabilistically derives two messages $\mu_1, \mu_2 \in \{0,1\}^k$, (from m and a random tape ω), computes $e_i \leftarrow p(\mu_i)$ and then the e_i-th root of y_i modulo n_i, for $i = 1, 2$, using the factorization of the moduli:

$$\{s_1 \leftarrow y_1^{1/e_1} \bmod n_1, s_2 \leftarrow y_2^{1/e_2} \bmod n_2\}$$

- To verify a given $\{m, \omega, s_1, s_2\}$, the verifier computes μ_1 and μ_2, then checks that $s_i^{p(\mu_i)} \bmod n_i \stackrel{?}{=} y_i$, for $i = 1, 2$.

To prevent forgeries, a new message must involve a new exponent, either e_1 or e_2, which never occurred in the signatures provided by the signing oracle. Therefore, a first requirement is that μ_1 and μ_2 define at most one message m, but only if they have been correctly constructed. Thus, some redundancy is furthermore required.

We thus suggest the following derivation, to get μ_1 and μ_2 from $m \in \{0,1\}^{k/2}$ (we assume k to be even): one chooses two random elements $a, b \in \{0,1\}^{k/2}$, then $\mu_1 = (m \oplus a)||(m \oplus b)$ and $\mu_2 = a||b$.

Clearly, given μ_1 and μ_2, one gets back $M = \mu_1 \oplus \mu_2$, which provides a valid message if and only if the redundancy holds: $\overline{M} = \underline{M}$, where $\overline{S}$ and $\underline{S}$ denote the two $k/2$-bit halves of a k-bit string S, the most significant and the least significant parts respectively.

5.5 Existential Forgeries

Let us show that existential forgery of the twin scheme, with above derivation process, leads to a new solution of the Flexible RSA Problem:

LEMMA 1. *After q queries to the signing oracle, the probability that there exist a new message m and values a, b,*

which lead to $\mu_1 = (m \oplus a)||(m \oplus b)$ *and* $\mu_2 = a||b$*, such that both* $e_1 = p(\mu_1)$ *and* $e_2 = p(\mu_2)$ *already occurred in the signatures provided by the signing oracle is less than* $q^2/2^{k/2}$.

PROOF. Let $\{m_i, a_i, b_i, s_{1,i}, s_{2,i}\}$ denote the answers of the signing oracle. Using the injectivity of p, the existence of such m, a and b means that there exist indices i and j for which

$$\begin{aligned}(m \oplus a)||(m \oplus b) = \mu_1 &= \mu_{1,i} = (m_i \oplus a_i)||(m_i \oplus b_i)\\ a||b = \mu_2 &= \mu_{2,j} = a_j||b_j.\end{aligned}$$

Then

$$a \oplus b = (m \oplus a) \oplus (m \oplus b) = (m_i \oplus a_i) \oplus (m_i \oplus b_i) = a_i \oplus b_i,$$

and

$$a \oplus b = a_j \oplus b_j.$$

Therefore, for a $j > i$ (the case $i > j$ is similar), the new random elements a_j, b_j must satisfy $a_j \oplus b_j = a_i \oplus b_i$. Since it is randomly chosen by the signer, the probability that this occurs for some $i < j$ is less than $(j-1)/2^{k/2}$.

Altogether, the probability that for some j there exists some $i < j$ which satisfies the above equality is less that $q^2/2 \times 2^{-k/2}$. By symmetry, we obtain the same result if we exchange i and j.

The probability that both exponents already appeared is consequently smaller than $q^2/2^{k/2}$. □

To prevent adaptively chosen-message attacks, we need no trapdoor property for p, nor random oracle assumption either. We simply give the factorization of one modulus to the simulator, which can use any pre-computed exponentiation with any new message, as when chameleon functions are used [8].

5.6 Adaptively Chosen-Message Attacks

Indeed, to prevent adaptively chosen-message attacks, one just needs to describe a simulator; our simulator works as follows:

- The simulator is first given the moduli n_1, n_2 and the elements $y_1 \in \mathbb{Z}^*_{n_1}$, $y_2 \in \mathbb{Z}^*_{n_2}$, as well as the factorization of n_γ, where γ is randomly chosen in $\{1, 2\}$. To simplify notations we assume that $\gamma = 1$. And the following works without loss of generality since the derivation of μ_1 and μ_2 is perfectly symmetric: they are randomly distributed, but satisfy $\mu_1 \oplus \mu_2 = m||m$ (it is a perfect secret sharing).
- The simulator randomly generates q values $e_{2,j} \leftarrow p(\mu_{2,j})$, with randomly chosen $\mu_{2,j} \in_R \{0,1\}^k$ for $j = 1, \ldots, q$ and computes

  $$z \leftarrow y_2^{\prod_{j=1,\ldots,q} e_{2,j}} \bmod n_2.$$

  The new public key for the signature scheme is the following: the moduli n_1, n_2 with the elements y_1, z in $\mathbb{Z}^*_{n_1}$ and $\mathbb{Z}^*_{n_2}$ respectively.
- For the j-th signed message m, the simulator first gets $(a||b) \leftarrow (m||m) \oplus \mu_{2,j}$. It therefore computes $\mu_1 \leftarrow a||b$, and thus $\mu_2 \leftarrow \mu_{2,j} = (m \oplus a)||(m \oplus b)$.

  Then, it knows $s_2 = y_2^{\prod_{i \neq j} e_{2,i}} \bmod n_2$, and computes s_1 using the factorization of n_1.

Such a simulator can simulate up to q signatures, which leads to the following theorem.

THEOREM 3. *Let us consider an adversary against the twin-GHR scheme who succeeds in producing an existential forgery, with probability greater than* ε*, after* q *adaptive queries to the signing oracle in time* t*, then the Flexible RSA Problem can be solved with probability greater than* ε' *within a time bound* t'*, where*

$$\varepsilon' = \frac{1}{2}\left(\varepsilon - \frac{q^2}{2^{k/2}}\right) \quad \text{and} \quad t' = t + \mathcal{O}(q \times \ell^2 \times k).$$

PROOF. Note that the above bounds are almost optimal since $\varepsilon' \cong \varepsilon/2$ and $t' \cong 2t$. Indeed, the time needed to produce an existential forgery after q signature queries is already in $\mathcal{O}(q \times (|n_1|^2 + |n_2|^2)k)$. To evaluate the success probability, q is less than say 2^{40}, but k may be taken greater than 160 bits (and even much more).

To conclude the proof, one just needs to address the random choice of γ. As we have seen in Lemma 1, with probability greater than $\varepsilon - q^2/2^{k/2}$, one of the exponents in the forgery never appeared before. Since γ is randomly chosen and the view of the simulation is perfectly independent of this choice, with probability of one half, $e = e_{\bar{\gamma}}$ is new. Let us follow our assumption that $\gamma = 1$, then

$$s^e = s_2^e = z = y_2^\pi \bmod n_2,$$

where $\pi = \prod_{j=1,\ldots,q} e_{2,j}$. Since e is new, it is relatively prime with π, and therefore, there exist u and v such that $ue + v\pi = 1$: let us define $x = y_2^u s^v \bmod n_2$,

$$x^e = (y_2^u s^v)^e = y_2^{1-v\pi} s^{ev} = y_2 (y_2^\pi)^{-v} (s^e)^v = y_2 \bmod n_2.$$

We thus obtain an e-th root of the given y_2 modulo n_2, for a new prime e. □

5.7 More Signatures

One may remark that the length of the messages we can sign with above construction is limited to $k/2$ bits, because of the required redundancy. But one can increase the size, by signing three derived messages: in order to sign $m \in \{0,1\}^k$, one chooses two random elements $a, b \in \{0,1\}^{k/2}$ (we still assume k to be even), and signs with different moduli

$$\begin{aligned}\mu_1 &= m \oplus (a||b)\\ \mu_2 &= a||b\\ \mu_3 &= m \oplus (b||a).\end{aligned}$$

6. CONCLUSION AND FURTHER RESEARCH

We proposed an alternative to the well-known hash-and-sign paradigm, based on the simple idea of signing twice (or more) identical or related short messages. We believe that our first investigations show that this is a promising strategy, deserving further study.

A number of interesting questions remain open. First, from the efficiency point of view, which is a frequent concern, we are aware that the current proposals do not deal with either the computational cost, or the communication load, in an efficient way. Thus, for example, can the number of fields in a twin DSA be reduced from four ($\{c, d\}$ and $\{c', d'\}$) to three or less? Can we also suppress some fields in the twin-GHR, or sign k-bit long messages with only two signatures?

Finally, can an increase in the number of signatures (*e.g.* three instead of two) yield better security bounds?

7. REFERENCES

[1] M. Abe and T. Okamoto. A Signature Scheme with Message Recovery as Secure as Discrete Logarithm. In *Asiacrypt '99*, LNCS 1716. Springer-Verlag, Berlin, 1999.

[2] N. Barić and B. Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes without Trees. In *Eurocrypt '97*, LNCS 1233, pages 480–484. Springer-Verlag, Berlin, 1997.

[3] M. Bellare and P. Rogaway. Random Oracles Are Practical: a Paradigm for Designing Efficient Protocols. In *Proc. of the 1st CCS*, pages 62–73. ACM Press, New York, 1993.

[4] J.-S. Coron and D. Naccache. Security Analysis of the Gennaro-Halevi-Rabin Signature Scheme. In *Eurocrypt '99*, LNCS 1592, pages 91–101. Springer-Verlag, Berlin, 1999.

[5] R. Cramer and V. Shoup. Signature Scheme based on the Strong RSA Assumption. In *Proc. of the 6th CCS*, pages 46–51. ACM Press, New York, 1999.

[6] T. El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, IT–31(4):469–472, July 1985.

[7] E. Fujisaki and T. Okamoto. Statistical Zero Knowledge Protocols to Prove Modular Polynomial Relations. In *Crypto '97*, LNCS 1294, pages 16–30. Springer-Verlag, Berlin, 1997.

[8] R. Gennaro, S. Halevi, and T. Rabin. Secure Hash-and-Sign Signature Without the Random Oracle. In *Eurocrypt '99*, LNCS 1592, pages 123–139. Springer-Verlag, Berlin, 1999.

[9] S. Goldwasser, S. Micali, and R. Rivest. A Digital Signature Scheme Secure Against Adaptative Chosen-Message Attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[10] IEEE P1363. Standard Specifications for Public Key Cryptography. Available from `http://grouper.ieee.org/groups/1363`, August 1998.

[11] M. Jakobsson and C. P. Schnorr. Security of Discrete Logarithm Cryptosystems in the Random Oracle Model and Generic Model. Available from `http://www.bell-labs.com/~markusj`, 1998.

[12] H. Krawczyk and T. Rabin. Chameleon Hashing and Signatures. In *Proc. of NDSS '2000*. Internet Society, 2000.

[13] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Available from `http://www.cacr.math.uwaterloo.ca/hac/`.

[14] D. M'Raïhi and D. Naccache. Batch Exponentiation – A Fast DLP-based Signature Generation Strategy. In *Proc. of the 3rd CCS*, pages 58–61. ACM Press, New York, 1996.

[15] V. I. Nechaev. Complexity of a Determinate Algorithm for the Discrete Logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

[16] NIST. Digital Signature Standard (DSS). Federal Information Processing Standards PUBlication 186, November 1994.

[17] K. Nyberg and R. A. Rueppel. Message Recovery for Signature Schemes Based on the Discrete Logarithm Problem. In *Eurocrypt '94*, LNCS 950, pages 182–193. Springer-Verlag, Berlin, 1995.

[18] J. M. Pollard. Monte Carlo Methods for Index Computation (mod p). *Mathematics of Computation*, 32(143):918–924, July 1978.

[19] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[20] C. P. Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[21] V. Shoup. Lower Bounds for Discrete Logarithms and Related Problems. In *Eurocrypt '97*, LNCS 1233, pages 256–266. Springer-Verlag, Berlin, 1997.

APPENDIX

A. TWIN SIGNATURES WITH MESSAGE RECOVERY

In this appendix, we describe a twin version of the Nyberg-Rueppel scheme [17] which provides message recovery. Keeping the notations of section 4.1:

1. Generate a random number u, $1 \leq u < r$.
2. Compute $c \leftarrow \sigma(u) + m \bmod r$. If $c = 0$ go to step 1.
3. Compute an integer $d \leftarrow u - cx \bmod r$.
4. Output the pair $\{c, d\}$ as the signature.

In the above, f is what is called in [10] a *message with appendix*. It simply means that it has an adequate redundancy. The corresponding verification is performed by the following (generic) steps:

1. If $c \notin [1, r-1]$ or $d \notin [0, r-1]$, output invalid and stop.
2. Obtain $\sigma(d + cx)$ from the oracle and compute $\gamma \leftarrow \sigma(d + cx) \bmod r$.
3. Check the redundancy of $m \leftarrow c - \gamma \bmod r$. If incorrect output invalid and stop; otherwise output the reconstructed message m, output valid and stop.

In the twin setting, signature generation is alike but is performed twice, so as to output two distinct signatures. However, no redundancy is needed. The verifier simply checks that the signatures are distinct and outputs two successive versions of the message, say m and m'. It returns valid if $m \stackrel{?}{=} m'$ and invalid otherwise. The security proof is sketched here, we leave the discussion of adaptive attacks to the reader.

We keep the notations and assumptions of section 4 and let $\mathcal{A}$ be a generic attacker over Γ which outputs, on input $\{\sigma(1), \sigma(x)\}$, two signature pairs $\{c, d\}$, $\{c', d'\}$ and runs the verifying algorithm that produces from these signatures two messages m, m' and checks whether they are equal. We wish to show that, if $x \in \Gamma$ and an encoding σ are chosen at random, then the probability that $m = m'$ is $\mathcal{O}(tn^2/r)$.

As before, we restrict our attention to behaviors of the full algorithm corresponding to safe sequences $\{z_1, \cdots, z_n, y\}$.

We let P, Q be the polynomials $d + cX$ and $d' + c'X$. We first consider the case where either P or Q does not appear in the F_i list before the signatures are produced. If this happens for P, then, P is included in the F_i list at signature verification and the corresponding answer of the oracle is a random number z_i. Since m is computed as $c - z_i \bmod r$, the probability that $m = m'$ is bounded by t/r. A similar bound holds for Q.

We now assume that both P and Q appear in the F_i list before $\mathcal{A}$ outputs its signatures. We let i denote the first index such that $F_i = P$ and j the first index such that $F_j = Q$. Note that both F_i and F_j are unmarked (as defined in section 3.1). If $i = j$, then we obtain that $c = c'$ and $d = d'$. From this, it follows that the signatures are not distinct.

As in section 4, we are left with the case where $i \neq j$ and we define $\Omega_{i,j}$, $i < j$, to be the set of safe sequences producing two signatures such that the polynomials P, Q, defined as above appear for the first time before the algorithm outputs the signatures, as F_i and F_j. We show that, for any fixed value $w = \{z_1, \cdots, z_{j-1}\}$, $\Omega_{i,j} \cap \hat{w}$ has probability $\leq t/r$, where $\hat{w}$ is defined as above. Since we have $m = c - z_i \bmod r$ and $m' = c' - z_j \bmod r$, we obtain $z_j = c' - c + z_i \bmod r$, from which the upper bound follows. From this bound, we obtain that the probability of $\Omega_{i,j}$ is at most t/r and, taking the union of the various $\Omega_{i,j}$s, we conclude that the probability to obtain a valid twin signature is at most $\mathcal{O}(tn^2/r)$.

B. THE CHOICE OF FUNCTION P

B.1 A Candidate

The following is a natural candidate:

$$\begin{aligned} p : \{0,1\}^k &\rightarrow \mathcal{P} \\ m &\mapsto \texttt{nextprime}(m \times 2^\tau) \end{aligned}$$

where τ is suitably chosen to guarantee the existence of a prime in any set $[m \times 2^\tau; (m+1) \times 2^\tau[$, for $m < 2^k$.

Note that the deterministic property of `nextprime` is not mandatory, one just needs it to be injective. But then, the preimage must be easily recoverable from the prime: the exponent is sent as the signature, from which one checks the primality and extracts the message (message-recovery).

B.2 Analysis

It is clear that any generator of random primes, using m as a seed, can be considered as a candidate for p. The function proposed above is derived from a technique for accelerating prime generation called *incremental search* (e.g. [13], page 148).

1. Input: an odd k-bit number n_0 (derived from m)
2. Test the s numbers n_0, $n_0 + 2$, ..., $n_0 + 2(s-1)$ for primality

Under reasonable number-theoretic assumptions, if $s = c \cdot \ln 2^k$, the probability of failure of this technique is smaller than $2e^{-2c}$, for large k.

Using our notations, in such a way that there exists at least a prime in any set $[m \times 2^\tau, (m+1) \times 2^\tau[$, but with probability smaller than 2^{-80}, we obtain from above formulae that $c \cong 40$, and $2^\tau \geq 40 \ln 2^{k+\tau+1}$. Therefore, a suitable candidate is $\tau \cong 5 \log_2 k$, and less than $20k$ primality tests have to be performed.

B.3 Extensions

B.3.1 Collision-resistance:

To sign large messages (at the cost of extra assumptions), one can of course use any collision-resistant hash-function h before signing (using the classical hash-and-sign technique). Clearly, the new function $m \mapsto p(h(m))$ is not mathematically injective, but just computationally injective (which is equivalent to collision-resistance), which is enough for the proof.

B.3.2 Division intractability:

If one wants to improve efficiency, using the division-intractability conjecture proposed in [8], any function that outputs k-bit strings can be used instead of p. More precisely:

Definition (Division Intractability). *A function H is said (n, ν, τ)-division intractable if any adversary which runs in time τ cannot find, with probability greater than ν, a set of elements a_1, ..., a_n and b such that $H(b)$ divides the product of all the $H(a_i)$.*

As above, that function p would not be injective, but just collision-resistant, which is enough to prove the following:

THEOREM 4. *Let us consider the twin-GHR scheme where p is any (q, ε, t)-division-intractable hash function. Let us assume that an adversary $\mathcal{A}$ succeeds in producing an existential forgery under an adaptively chosen-message attack within time t and with probability greater than ε, after q queries to the signing oracle. Then one can either contradict the division-intractability assumption or solve the Flexible RSA Problem with probability greater than ε' within a time bound t', where*

$$\varepsilon' = \frac{1}{2}\left(\varepsilon - \frac{q^2}{2^{k/2}}\right) \quad \text{and} \quad t' = t + \mathcal{O}(q \times \ell^2 \times k).$$

Batch Exponentiation
- A Fast DLP-based signature generation strategy -

David M'RAÏHI **David NACCACHE**

GEMPLUS, Crypto Team, 1 place de Méditerranée
F-95208, Sarcelles CEDEX, FRANCE
[100145.2261 and 100142.3240]@compuserve.com

Abstract : The signature generation phase of most DLP-based signature schemes (*for instance* **Schnorr**[10], **El-Gamal**[4] or the newly standardized **D.S.A.**[3]) includes the time-consuming computation of $r = g^k \bmod p$ where k is random.

This paper introduces a new computational strategy that can apply in this particular context :

A *batch exponentiation* technique which allows the generation of large sets of exponentials without introducing any bias between the *k*s (that is, the signer can batch-compute the exponentials corresponding to arbitrarily imposed powers -for instance by an external random number generator). Our method offers real improvements over the prior art with various time and memory trade-offs.

1. Introduction

In many DLP-based signature schemes[1] the signer performs the operation $r = g^k \bmod p$ where k is random. As the signer is often the " weak party " in the signature protocol, several authors tried to accelerate the exponentiation by pre-computing values [1], [5] or sub-contracting a part of the exponentiation workload to the verifier [6] (provided that a set of precautions is taken into consideration). Except the fact that some of these algorithms were broken [7], [8], extra memory storage is frequently an unrealistic assumption.

In this paper, we investigate a strategy for improving the generation of r : the method (providing improvements ranging from 42% to 85% over the *square-&-multiply* algorithm) can apply to the batch generation of fixed-g-based signatures without introducing any bias into the exponents (that is, we assume that the *k*s are **imposed** to the signer by some random source). We assume that no pre-computation is allowed other than what needed to execute similar size basic *square-& multiply*. The new method may as well open the way to interesting developments for accelerating the computation of discrete logarithms.

CCS '96, New Delhi, India
© 1996 ACM 0-89791-829-0/96/03..$3.50

2. Unbiased Batch-Exponentiation

The simultaneous signature of many messages by a shop terminal or the processing of electronic documents by an administration frequently involves massive computations consisting in the repetition of small operations. The re-combination of these computations for minimizing the computational effort is thus an interesting research direction (see for instance [10]).
The batch exponentiation technique proposed hereafter is built around the following observation : since the exponents are random, the uniform distribution of ones in different exponents is expected to have some matching patterns. Considering this fact, our batch-exponentiation strategy consists in minimizing the signer's workload **by exponentiating the intersection separately and resetting the corresponding bits** in the initial exponents.

2.1 Parallel *square-&-multiply (straightforward)*

Let n be the size of the exponents and N the number of exponentials to be computed. The usual method to generate the rs consists in calculating successive squares of g and performing the required multiplications selected by the bits of each k_i (about $n/2$ multiplications).

Let $S = \{k_i \mid i = 1,\ldots,N\}$ be a set of random powers. The corresponding set of exponentials

$$R = \{r_i \mid i = 1,\ldots,N\}$$

can be computed by performing the successive squares of g only once. Thus, the total computational effort mainly relies on the average number of multiplications, depending on the Hamming weight of each random exponent.

The algorithm **PSM**(S,R), of complexity

$$E(N) = N(n/2 - 1) + (n-1)$$

using $N + 1$ registers, is :

```
for i ← 1 to N    r_i ← 1
for j ← 0 to n−1
 for i ← 1 to N   if k_i[j] = 1 then r_i ← (r_i * g) mod p
 g ← g * g mod p
```

$$\text{where } k_i = \sum_{j=0}^{n-1} k_i[j]2^j .$$

2.2 The Basic Strategy

Denoting S a set of N random powers k, the strategy to generate the related exponentials is the following :

❶ Cut S into $T = \lfloor N/L \rfloor$ sets $s_h = \{k_j\}_{1 \le j \le L}$ where $L \le 5$[2]

❷ Let $P(s_h) = \bigcup_{i=1}^{2^L - 1} s_{h,i}$ where $s_{h,i}$ are s_h's subsets with $s_{h,i} \neq \emptyset$

[1] for instance [3], [4] and [10]

[2] L value depends on the exponent length; further explanation will be found in 3.2

❸ For $h \leftarrow 1$ to T

① For $i \leftarrow 1$ to $2^L - 1$ $\quad c_{h,i} = \bigotimes_{k_j \in s_{h,i}} k_j$

② For $C \leftarrow L-1$ to 1

For $i \leftarrow 1$ to $2^L - 1$

if $Card(s_{h,i}) = C$ then

$$c_{h,i} = c_{h,i} - \sum_{j \neq i, s_{h,i} \subset s_{h,j}} c_{h,j}$$

③ **PSM**$(\{c_{h,i}\}, R)$

④ For $i \leftarrow 1$ to L

$$g^{k_i} = \prod_{k_i \in s_{h,j}} r_j \quad \text{where } r_j \in R$$

The idea is to operate on each subset, reseting the common bits of the powers, and compute the exponentials together to save both squarrings and multiplications. The following section will describe the method for a reduced only two-power set.

2.3 Exponent Combination Method

Hereafter, we only consider the number of multiplications required to compute the r_i's assuming that it is possible to calculate the squares only once by the previously described parallel method.

Denoting

$$a = \sum_i a_i 2^i, \; b = \sum_i b_i 2^i$$

and assuming that $g^a \bmod p$ and $g^b \bmod p$ are to be computed, let $c = \sum_i a_i b_i 2^i$.

If a and b are randomly chosen, one should expect that :

$$\sum_i a_i \equiv \sum b_i \equiv \frac{n}{2} \text{ and } \sum_i c_i \equiv \frac{n}{4}.$$

Given the fact that :

$$w(a-c) + w(b-c) + w(c) \leq w(a) + w(b)$$

(where w denotes the Hamming weight), our strategy consists in computing :

$$\begin{cases} G_a = g^{a \oplus c} \bmod p \\ G_b = g^{b \oplus c} \bmod p \\ G_c = g^c \bmod p \end{cases}$$

to obtain

$$\begin{cases} r_a = G_a G_c \bmod p = g^a \bmod p \\ r_b = G_b G_c \bmod p = g^b \bmod p \end{cases}$$

The gain for a set of N signatures is therefore statistically $N(n/4 - 1)$ multiplications, which tends to **25%** of the total multiplications required to generate a set of N signatures with the parallel *square-&-multiply*, if we simply apply this strategy to all signatures grouped by pairs as illustrated in the following table :

operations	**PSM**	*Exponent Combination*
r_a	$\equiv n/2-1$	$\equiv n/2-n/4-1$
r_b	$\equiv n/2-1$	$\equiv n/2-n/4-1$
g^c	none	$\equiv n/4-1$
Total	$\equiv n-2$	$\equiv 3n/4-3$

Table 1 : PSM and batch exponentiation performance

The computational effort required to generate the set

$$G = \left\{ g^{k_i} \bmod p \mid i = 1, \ldots, N \right\}$$

is about $N/2(3n/4-3)) + n/N$ but implies to use $3N/2$ *size(p)*-bit registers which may not be practical in some situations. In the following we will present an optimization of our batch strategy and achieve comparison with Brickell, Gordon, McCurley and Wilson algorithm in [1], exhibiting when our strategy is more convenient and suitable.

3. Improvement and Performance

3.1 *BGCW* algorithm

The precomputation technique proposed by Brickell and al. in [1] is based on the following observation : in [11] it was proposed to precompute the set of g^{2^r} to reduce the computational effort increasing the storage amount. There is no reason to consider powers of 2.
The idea is to find a decomposition

$$k = \sum_{i=0}^{m-1} a_i x_i,$$

where $0 \leq a_i \leq h$ for $0 \leq i \leq m$, then we can compute

$$g^k = \prod_{d=1}^{h} c_d^d,$$

where $c_d = \prod_{a_i = d} g^{x_i}$.

The algorithm directly derived from this achieves the computation of g^k with only $m+h-2$ multiplications, but required that the values of $g^{x_i} \bmod p$ have been previously stored. To give an overview of the performances of this algorithm, one can remark that to generate a g^k with a 160-bit k, say for a Schnorr or DSS signature, using a base 16 representation for the numbers, the *BGCW* algorithm produces g^k at the cost of only 50 multiplications but requires also the storage of 40 n-bit numbers. Considering a base 32 notation, one can save a few storage (about 2 Kbytes rather than 2.5 previously) but the number of multiplications grows to 60. An improvement based on the notion of a *basic digit set* improves drastically the speed performance since the number of multiplications falls to about 36 but implies the storage of more than 200 numbers. The main drawback of the method is clearly the minimum storage capacity needed to achieve an exponentiation.

3.2 Optimizing the exponent combination

The total number of computations required to produce the set of g^ks will depend mainly on the multiplications to be calculated since the squarings are done once for all.
Considering that we want to group α n-bit exponents from a large set of N values together rather than only joining them by pairs, the

main issue is to find the best combination strategy to reduce the multiplications to be done.
The number of multiplications per g^k is in average

$$M(n,\alpha)=\frac{n(2^\alpha-1)}{\alpha 2^\alpha}+2^{\alpha-1}-1$$

and the squaring effort can be divided between all the *k*s.

The analysis of the function representation (see Annex 1) for usual exponent lengths (160-bit and 512-bit) give integer solutions which minimize this quantity. The number of registers required to achieve the computation increasing with the number of elements in the set (since to compute squarings once we must calculate all the g^k s together), the best values appear to be 4 for 160-bit values and 5 for 512-bit values. The final choice for a dedicated computation relies mainly on the memory capacity of the machine which generates the g^k s.

3.3 Performance Overview

The various strategies achieving several time and memory trade-offs are to be considered. The Tables 1 and 2 give the different trade-offs for 160-bit (Schnorr, DSS) and 512-bit (El- Gamal, Brickell-McCurley) exponents, considering that p is a 512-bit prime modulus.

Set Size	*Subset Size*	*Storage (Bytes)*	*#Multiplications per computation*
2	2	316	141
3	3	652	103
4	2	568	101
4	4	1,324	84.5
12	3	2,416	63
12	4	3,844	57.83
36	3	7,120	54.11
36	4	11,404	48.94
108	4	34,084	45.98

Table 2 : performances with 160-bit exponent

Set Size	*Subset Size*	*Storage (Bytes)*	*#Multiplications per computation*
2	2	448	449
3	3	960	323
4	4	1,984	255
5	5	4,032	216.6
60	3	17,984	160.87
60	4	28,864	135.53
60	5	47,680	122.73
200	5	158,784	116.76

Table 3 : performances with 512-bit exponent

Compare to the *Square-&-Multiply* and *BGCW* algorithms, the *Batch Exponentiation* strategy provides nice improvements to save computation and memory. The pairs grouping provides a 42 % gain on the total computation at the cost of only 2 *n*-bit registers, while the *BGCW* implies at least a 2 Kbytes storage. The technique is perfectly suited to low-memory environment where memory cost is high; furthermore, the exponent re-combination is easy to perform on any machine. On the other hand, BGCW algorithm is very efficient when at least a few Kbytes of permanent memory are available.

An adaptation of *Batch Exponentiation* tailored for high-speed transmission (see Annex 2) even provides greater improvements by sub-contracting the squaring effort to an external device assuming nothing on his security.

4. Conclusion, Extensions and Open Questions

We presented a strategy which can accelerate the generation of DLP-based signatures. The main characteristics of the batch-exponentiation technique presented in this article are summarized in the following table.

scheme⇨ effort⇩	*Batch (memory)*	*Batch (time)*
Schnorr	141 *N*	45.98 *N*
El-Gamal	449 *N*	116.76 *N*

Table 4 : exponentiation performance

Several open questions appear interesting to explore to further improve the proposed strategies :

- For a power $\kappa \in \mathcal{L}_q$, try to find a such that $k' = a\,q + k$ where the hamming weight of k' is significatively small. Since computations are done modulo q, this transformation of k does not have any impact on the result itself but may well reduce the computation workload.
- Find an algorithm such that the construction of the subsets is optimal, that is the ordering of the *k*s results in as few computations as possible.

Acknowledgments

The authors would like to thank Jacques Stern for his pertinent remarks and gentle support. Furthermore, during the CRYPTO'95 Rump Session Yokua Tsuruoka pointed out that he described a similar method in [12] at JWS'93 . We didn't know anything about this paper so that we can honestly consider Tsuruoka's paper as an independent prior discovery of the same algorithm.

References

[1] **E. Brickell, D. Gordon and K. McCurley,** *Fast exponentiation with precomputation*, technical report no. SAND91-1836C, Sandia National Laboratories, Albuquerque, New-Mexico, October 1991.

[2] **A. Fiat,** *Batch RSA*, Advances in cryptology: Proceedings of Crypto'89, LNCS, Springer-Verlag, 435, pp. 175-185.

[3] FIPS PUB 186, February 1, 1993, *Digital Signature Standard.*

[4] **T. El-Gamal,** *A public-key cryptosystem and a signature scheme based on discrete logarithms*, IEEE TIT, vol. IT-31:4, pp 469-472, 1985.

[5] **D. Naccache, D. M'raïhi, S. Vaudenay and D. Raphaeli,** *Can DSA be improved ? - Complexity Trade-Offs with the Digital Signature Standard*, Advances in cryptology: Proceedings of Eurocrypt'94, Perugia, LNCS 950, pp. 77-85, Springer-Verlag, 1995.

[6] **J.-J. Quisquater and M. de Soete**, *Speeding up smart-card RSA computation with Insecure Coprocessors*, Proceedings of Smart Cards 2000, 1989, pp. 191-197.

[7] **P.J.N de Rooij**, *On The Security of the Schnorr Scheme using Preprocessing*, Advances in cryptology: Proceedings of Eurocrypt'91, Brighton, LNCS 547, pp. 71-80, Springer-Verlag, 1991.

[8] **P.J.N de Rooij**, *On Schnorr's Preprocessing for Digital Signature Schemes*, Advances in cryptology: Proceedings of Eurocrypt'93, Lofthus, LNCS 765, pp. 435-439, Springer-Verlag, 1994.

[9] **Ryo-Fuji-Hara**, *Cipher Algorithms and Computational Complexity*, Bit 17 (1985), 954-959.

[10] **C. Schnorr**, *Efficient Identification and Signatures for Smart-Cards*, Advances in cryptology: Proceedings of Eurocrypt'89, Berlin, LNCS 435, pp. 239-252, Springer-Verlag, 1990.

[11] **J. Stern and S.Vaudenay**, *Personal Communication*, 1994.

[12] **Y. Tsuruoka**, *A Fast Algorithm on Addition Sequence*, JW-ISC'93, 1993

Appendix 1 : $M(n,\alpha)=\frac{n(2^{\alpha}-1)}{\alpha 2^{\alpha}}+2^{\alpha-1}-1$

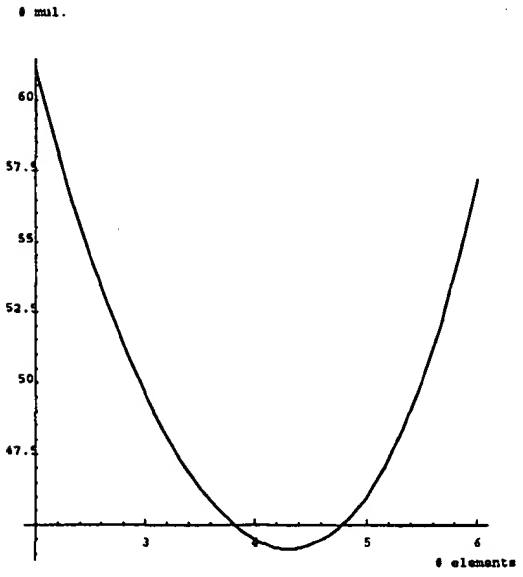


Figure 1 : 160-bit exponent - Best integer solution = 4

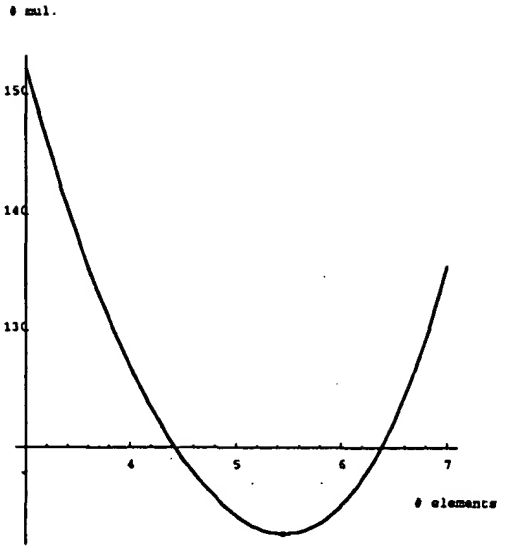


Figure 2 : 512-bit exponent - Best integer solution = 5

Appendix 2 : Sub-contracting squarings with high-speed transmission

Using a high-speed transmission interface such that of a PCMCIA card, one can subcontract the square computation and rely on the device with the same level of security.

The strategy remains the same, grouping the exponents and computing the g^k's, except that the squaring are computed by a genuine device, assuming nothing on his tamper-resistance. The device that shall compute g^k values has a certificate C on the set of $\left\{g^{2^i} \bmod p \mid i \le size(p)\right\}$ such as an iterative hashing of the whole set.

Denoting ***Sender*** the computing device in charge of the squaring effort and ***Receiver*** the exponentiation machine, the protocol is the following :

Sender		***Receiver***
$s = g$		$l = \varnothing$
For $i = 0$ to $n-1$ Send s	⟶	Receive s (use if needed)
		$s = s^2 \bmod p$
		$l = SHA(l\|s)$
		If $C = l$ accept

The VLDB Journal (2001) 10: 120–132 / Digital Object Identifier (DOI) 10.1007/s007780100047

PicoDBMS: Scaling down database techniques for the smartcard

Philippe Pucheral[1], Luc Bouganim[1], Patrick Valduriez[2], Christophe Bobineau[1]

[1] University of Versailles, PRiSM Laboratory, Versailles, France;
E-mail: {philippe.pucheral;luc.bouganim;christophe.bobineau}@prism.uvsq.fr
[2] University Paris 6, LIP6 Laboratory, Paris, France; E-mail: patrick.valduriez@lip6.fr

Edited by A. El Abbadi, G. Schlageter, K.-Y. Whang. Received: 15 October 2000 / Accepted: 15 April 2001
Published online: 23 July 2001 –

Abstract. Smartcards are the most secure portable computing device today. They have been used successfully in applications involving money, and proprietary and personal data (such as banking, healthcare, insurance, etc.). As smartcards get more powerful (with 32-bit CPU and more than 1 MB of stable memory in the next versions) and become multi-application, the need for database management arises. However, smartcards have severe hardware limitations (very slow write, very little RAM, constrained stable memory, no autonomy, etc.) which make traditional database technology irrelevant. The major problem is scaling down database techniques so they perform well under these limitations. In this paper, we give an in-depth analysis of this problem and propose a PicoDBMS solution based on highly compact data structures, query execution without RAM, and specific techniques for atomicity and durability. We show the effectiveness of our techniques through performance evaluation.

Key words: Smartcard applications – PicoDBMS – Storage model – Execution model – Query optimization – Atomicity – Durability

1 Introduction

Smartcards are the most secure portable computing device today. The first smartcard was developed by Bull for the French banking system in the 1980s to significantly reduce the losses associated with magnetic stripe credit card fraud. Since then, smartcards have been used successfully around the world in various applications involving money, proprietary data, and personal data (such as banking, pay-TV or GSM subscriber identification, loyalty, healthcare, insurance, etc.). While today's smartcards handle a single issuer-dependent application, the trend is toward multi-application smartcards[1]. Standards for multi-application support, like the JavaCard [36] and Microsoft's SmartCard for Windows [26], ensure that the card be universally accepted and be able to interact with several service providers. This should make smartcards one of the world's highest-volume markets for semiconductors [14].

As smartcards become more and more versatile, multi-application, and powerful (32-bit processor, more than 1 MB of stable storage), the need for database techniques arises. Let us consider a health card storing a complete medical folder including the holder's doctors, blood type, allergies, prescriptions, etc. The volume of data can be important and the queries fairly complex (select, join, aggregate). Sophisticated access rights management using views and aggregate functions are required to preserve the holder's data privacy. Transaction atomicity and durability are also needed to enforce data consistency. More generally, database management helps to separate data management code from application code, thereby simplifying and making application code smaller. Finally, new applications can be envisioned, like computing statistics on a large number of cards, in an asynchronous and distributed way. Supporting database management on the card itself rather than on an external device is the only way to achieve very high security, high availability (anywhere, anytime, on any terminal), and acceptable performance.

However, smartcards have severe hardware limitations which stem from the obvious constraints of small size (to fit on a flexible plastic card and to increase hardware security) and low cost (to be sold in large volumes). Today's microcontrollers contain a CPU, memory – including about 96 kB of ROM, 4 kB of RAM, and up to 128 kB of stable storage like EEPROM – and security modules [39]. EEPROM is used to store persistent information; it has very fast read time (60–100 ns) comparable to old-fashion RAM but very slow write time (more than 1 ms/word). Following Moore's law for processor and memory capacities, smartcards will get rapidly more powerful. Existing prototypes, like Gemplus's Pinocchio card [16], bypass the current memory bottleneck by connecting an additional chip of 2 MB of Flash memory to the microcontroller. Although a significant improvement over today's cards, this is still very restricted compared to other portable, less secure, devices such as Personal Digital Assistants (PDA). Furthermore, smartcards are not autonomous, i.e., have no independent power supply, thereby precluding asynchronous and disconnected processing.

[1] Everyone would probably enjoy carrying far fewer cards.

These limitations (tiny RAM, little stable storage, very costly write, and lack of autonomy) make traditional database techniques irrelevant. Typically, traditional DBMS exploit significant amounts of RAM and use caching and asynchronous I/Os to reduce disk access overhead as much as possible. With the extreme constraints of the smartcard, the major problem is scaling down database techniques. While there has been much excellent work on scaling up to deal with very large databases, e.g., using parallelism, scaling down has not received much attention by the database research community. However, scaling down in general is becoming very important for commodity computing and is quite difficult [18].

Some DBMS designs have addressed the problem of scaling down. Light versions of popular DBMS like Sybase Adaptive Server Anywhere [37], Oracle 8i Lite [30] or DB2 Everywhere [20] have been primarily designed for portable computers and PDA. They have a small footprint which they obtain by simplifying and componentizing the DBMS code. However, they use relatively high RAM and stable memory and do not address the more severe limitations of smartcards. ISOL's SQLJava Machine DBMS [13] is the first attempt towards a smartcard DBMS while SCQL [24], the standard for smartcard database language, emerges. While both designs are limited to single select, they exemplify the strong interest for dedicated smartcard DBMS.

In this paper, we address the problem of scaling down database techniques and propose the design of what we call a PicoDBMS. This work is done in the context of a new project with Bull Smart Cards and Terminals. The design has been made with smartcard applications in mind, but its scope extends as well to any ultra-light computer device based on a secured monolithic chip. This paper makes the following contributions:

- We analyze the requirements for a PicoDBMS based on a typical healthcare application and justify its minimal functionality.
- We give an in-depth analysis of the problem by considering the smartcard hardware trends and derive design principles for a PicoDBMS.
- We propose a new pointer-based storage model that integrates data and indices in a unique compact data structure.
- We propose query execution techniques which handle complex query plans (including joins and aggregates) with no RAM consumption.
- We propose transaction techniques for atomicity and durability that reduce the logging cost to its lowest bound and enable a smartcard to participate in distributed transactions.
- We show the effectiveness of each technique through performance evaluation.

This paper is an extended version of [7]. In particular, the section on transaction management is new. The paper is organized as follows. Section 2 illustrates the use of take-away databases in various classes of smartcard applications and presents in more detail the requirements of the health card application. Section 3 analyzes the smartcard hardware constraints and gives the problem definition. Sections 4–6 present and assess the PicoDBMS' storage model, query execution model, and transaction model, respectively. Section 7 concludes.

2 Smartcard applications

In this section, we discuss the major classes of emerging smartcard applications and their database requirements. Then, we illustrate these requirements in further detail with the health card application, which we will use as reference example in the rest of the paper.

2.1 Database management requirements

Table 1 summarizes the database management requirements of the following typical classes of smartcard applications:

- *Money and identification*: examples of such applications are credit cards, e-purse, SIM for GSM, phone cards, transportation cards. They are representative of today's applications, with very few data (typically the holder's identifier and some status information). Querying is not a concern and access rights are irrelevant since cards are protected by PIN-codes. Their unique database management requirement is update atomicity.
- *Downloadable databases*: these are predefined packages of confidential data (e.g., diplomatic, military or business information) that can be downloaded on the card – for example, before traveling – and be accessed from any terminal. Data availability and security are the major concerns here. The volume of data can be important and the queries complex. The data are typically read-only.
- *User environment*: the objective is to store in a smartcard an extended profile of the card's holder including, among others, data regarding the computing environment (PC's configuration, passwords, cookies, bookmarks, software licenses, etc.), an address book as well as an agenda. The user environment can thus be dynamically recovered from the profile on any terminal. Queries remain simple, as data are not related. However, some of the data are highly private and must be protected by sophisticated access rights (e.g., the card's holder may want to share a subset of her/his address book or bookmark list with a subset of persons). Transaction atomicity and durability are also required.
- *Personal folders*: personal folders may be of a different nature: scholastic, healthcare, car maintenance history, loyalty. They roughly share the same requirements, which we illustrate next with the healthcare example. Note that queries involving data issued from different folders can make sense. For instance, one may be interested in discovering associations between some disease and the scholastic level of the card holder. This raises the interesting issue of maintaining statistics on a population of cards or mining their content asynchronously.

2.2 The health card application

The health card is very representative of personal folder applications and has strong database requirements. Several countries (France, Germany, USA, Russia, Korea, etc.) are developing healthcare applications on smartcards [11]. The initial idea was to give to each citizen a smartcard containing her/his identification and insurance data. As smartcard storage capacity increases, the information stored in the card can be

Table 1. Typical applications' profiles

Applications	Volume	Select/project	Join	Group by / Distinct	Access rights / views	Atomicity	Durability	Statistics
Money & identification	tiny					✓		
Downloadable DB	high	✓	✓	✓				
User environment	medium	✓			✓	✓	✓	
Personal folder	high	✓	✓	✓	✓	✓	✓	✓

extended to the holder's doctors, emergency data (blood type, allergies, vaccination, etc.), surgical operations, prescriptions, insurance data and even links to heavier data (e.g., X-ray examination, scanner images, etc.) stored on hospital servers. Different users may query, modify, and create data in the holder's folder: the doctors who consult the patient's past records and prescribe drugs, the surgeons who perform exams and operations, the pharmacists who deliver drugs, the insurance agents who refund the patient, public organizations which maintain statistics or study the impact of drugs correlation in population samples, and finally the holder her/himself.

We can easily observe that: (i) the amount of data is significant (more in terms of cardinality than in terms of volume because most data can be encoded); (ii) queries can be rather complex (e.g., a doctor asks for the last antibiotics prescribed to the patient); (iii) sophisticated access rights management using views and aggregate functions are highly required (e.g., a statistical organization may access aggregate values only but not the raw data); (iv) atomicity must be preserved (e.g., when the pharmacist delivers drugs); and (v) durability is mandatory, without compromising data privacy (logged data stored outside the card must be protected).

One may wonder whether the holder's health data ought to be stored in a smartcard or in a centralized database. The benefit of distributing the healthcare database on smartcards is threefold. First, health data must be made highly available (anywhere, anytime, on any terminal, and without requiring a network connection). Second, storing sensitive data on a centralized server may damage privacy. Third, maintaining a centralized database is fairly complex due to the variety of data sources. Assuming the health data is stored in the smartcard, the next question is why the aforementioned database capabilities need to be hosted in the smartcard rather than the terminals. The answer is again availability (the data must be exploited on any terminal) and privacy. Regarding privacy, since the data must be confined in the chip, so must the query engine and the view manager. As the smartcard is the unique trusted part of the system, access rights and transaction management cannot be delegated to an untrusted terminal.

3 Problem formulation

In this section, we make clear the smartcard constraints in order to derive design rules for the PicoDBMS and state the problem. Our analysis is based on the characteristics of both existing smartcard products and current prototypes [16, 39], and thus, should be valid for a while. We also discuss how the main constraints of the smartcard will evolve in a near future.

3.1 Smartcard constraints

Current smartcards include in a monolithic chip, a 32 bits RISC processor at about 30 MIPS, memory modules (of about 96 kB of ROM, 4 kB of static RAM, and 128 kB of EEPROM), security components (to prevent tampering), and take their electrical energy from the terminal [39]. ROM is used to store the operating system, the JavaCard virtual machine, fixed data, and standard routines. RAM is used as working memory for maintaining an execution stack and calculating results. EEPROM is used to store persistent information. EEPROM has very fast read time (60–100 ns/word) comparable to old-fashion RAM, but a dramatically slow write time (more than 1 ms/word).

The main constraints of current smartcards are therefore: (i) the very limited storage capacity; (ii) the very slow write time in EEPROM; (iii) the extremely reduced size of the RAM; (iv) the lack of autonomy; and (v) a high security level that must be preserved in all situations. These constraints strongly distinguish smartcards from any other computing devices, including lightweight computers like PDA.

Let us now consider how hardware advances can impact on these constraints, in particular, memory size. Current smartcards rely on a well-established and slightly out-of-date hardware technology (0.35μm) in order to minimize the production cost (less than five dollars) and increase security [34]. Furthermore, up to now, there was no real need for large memories in smartcard applications such as the holder's identification. According to major smartcard providers, the market pressure generated by emerging large storage demanding applications will lead to a rapid increase of the smartcard storage capacity. This evolution is however constrained by the smartcard tiny die size fixed to 25 mm^2 in the ISO standard [23], which pushes for more integration. This limited size is due to security considerations (to minimize the risk of physical attack [5]) and practical constraints (e.g., the chip should not break when the smartcard is flexed). Another solution to relax the storage limit is to extend the smartcard storage capacity with external memory modules. This is being done by Gemplus which recently announced Pinocchio [16], a smartcard equipped with 2 MB of Flash memory linked to the microcontroller by a bus. Since hardware security can no longer be provided on this memory, its content must be either non-sensitive or encrypted.

Another important issue is the performance of stable memory. Possible alternatives to the EEPROM are Flash memory and Ferroelectric RAM (FeRAM) [15] (see Table 2 for performance comparisons). Flash is more compact than EEPROM and represents a good candidate for high capacity smartcards [16]. However, Flash banks need to be erased before writing, which is extremely slow. This makes Flash memory appropriate for applications with a high read/write ratio (e.g., address books). FeRAM is undoubtedly an interesting option for smartcards as read and write times are both fast. Although its theoretical foundation was set in the early 1950s, FeRAM is just emerging as an industrial solution. Therefore, FeRAM is expensive, less secure than EEPROM or Flash, and its integration with traditional technologies (such as CPUs) remains an

Table 2. Performance of stable memories for the smartcard

Memory type	EEPROM	FLASH	FeRAM
Read time (/word)	60 to 150 ns	70 to 200 ns	150 to 200 ns
Write time (/word)	1 to 5 ms	5 to 10 μs	150 to 200 ns
Erase time (/bank)	None	500 to 800 ms	None
Lifetime (*) (/cell)	10^5 write cycles	10^5 erase cycles	10^{10} to 10^{12} write cycles

* A memory cell can be overwritten a finite number of time.

issue. Thus FeRAM could be considered a serious alternative only in the very long term [15].

Given these considerations, we assume in this paper a smartcard with a reasonable stable storage area (a few megabytes of EEPROM[2]) and a small RAM area (some kilobytes). Indeed, there is no clear interest in having a large RAM area, given that the smartcard is not autonomous, thus precluding asynchronous write operations. Moreover, more RAM means less EEPROM as the chip size is limited.

3.2 Impact on the PicoDBMS architecture

We now analyze the impact of the smartcard constraints on the PicoDBMS architecture, thus justifying why traditional database techniques, and even lightweight DBMS techniques, are irrelevant. The smartcard's properties and their impact are:

- *Highly secure*: smartcard's hardware security makes it the ideal storage support for private data. The PicoDBMS must contribute to the data security by providing access right management and a view mechanism that allows complex view definitions (i.e., supporting data composition and aggregation). The PicoDBMS code must not present security holes due to the use of sophisticated algorithms[3].
- *Highly portable*: the smartcard is undoubtedly the most portable personal computer (the wallet computer). The data located on the smartcard are thus highly available. They are also highly vulnerable since the smartcard can be lost, stolen or accidentally destroyed. The main consequence is that durability cannot be enforced locally.
- *Limited storage resources*: despite the foreseen increase in storage capacity, the smartcard will remain the lightest representative of personal computers for a long time. This means that specific storage models and execution techniques must be devised to minimize the volume of persistent data (i.e., the database) and the memory consumption during execution. In addition, the functionalities of the PicoDBMS must be carefully selected and their implementation must be as light as possible. The lightest the PicoDBMS, the biggest the onboard database.
- *Stable storage is main memory*: smartcard stable memory provides the read speed and direct access granularity of a main memory. Thus, a PicoDBMS can be considered as a *main memory DBMS* (*MMDBMS*). However the dramatic cost of writes distinguishes a PicoDBMS from a traditional MMDBMS. This impacts on the storage and access methods of the PicoDBMS as well as the way transaction atomicity is achieved.
- *Non-autonomous*: compared to other computers, the smartcard has no independent power supply, thereby precluding disconnected and asynchronous processing. Thus, all transactions must be completed while the card is inserted in a terminal (unlike PDA, write operations cannot be cached in RAM and reported on stable storage asynchronously).

[2] Considering Flash instead of EEPROM will not change our conclusions. It will just exacerbate them.

[3] Most security holes are the results of software bugs [34].

3.3 Problem statement

To summarize, our goal is to design a PicoDBMS including the following components:

- *Storage manager*: manages the storage of the database and the associated indices.
- *Query manager*: processes query plans composed of select, project, join, and aggregates.
- *Transaction manager*: enforces the ACID properties and participates to distributed transactions.
- *Access right manager*: provides access rights on base data and on complex user-defined views.

Thus, the PicoDBMS hosted in the chip provides the minimal subset of functionality that is strictly needed to manage in a secure way the data shared by all onboard applications. Other components (e.g., the GUI, a sort operator, etc.) can be hosted in the terminal or be dynamically downloaded when needed, without threatening security. In the rest of this paper, we concentrate on the components which require non-traditional techniques (storage manager, query manager, and transaction manager) and ignore the access right manager for which traditional techniques can be used.

When designing the PicoDBMS's components, we must follow several design rules derived from the smartcard's properties:

- *Compactness rule*: minimize the size of data structures and the PicoDBMS code to cope with the limited stable memory area (a few megabytes).
- *RAM rule*: minimize the RAM usage given its extremely limited size (some kilobytes).
- *Write rule*: minimize write operations given their dramatic cost ($\approx$ 1 ms/word).
- *Read rule*: take advantage of the fast read operations ($\approx$ 100 ns/word).
- *Access rule*: take advantage of the low granularity and direct access capability of the stable memory for both read and write operations.
- *Security rule*: never externalize private data from the chip and minimize the algorithms' complexity to avoid security holes.

4 PicoDBMS storage model

In this section, following the design rules for a PicoDBMS, we discuss the storage issues and propose a very compact model based on a combination of flat storage, domain storage, and ring storage. We also evaluate the storage cost of our storage model.

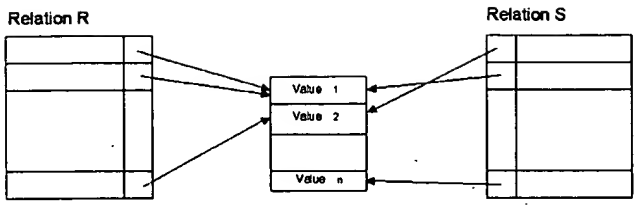


Fig. 1. Domain storage

4.1 Flat storage

The simplest way to organize data is *flat storage (FS)*, where tuples are stored sequentially and attribute values are embedded in the tuples. Although it does not impose it, the SCQL standard [24] considers FS as the reference storage model for smartcards. The main advantage of FS is access locality. However, in our context, FS has two main drawbacks:

- *Space consuming*: while normalization rules preclude attributes conjunction redundancy to occur, they do not avoid attribute value duplicates (e.g., the attribute *Doctor.Specialty* may contain many duplicates).
- *Inefficient*: in the absence of index structures, all operations are computed sequentially. While this is convenient for old fashion cards (some kilobytes of storage and a mono-relation select operator), this is no longer acceptable for future cards where storage capacity is likely to exceed 1 MB and queries can be rather complex.

Adding index structures to FS may solve the second problem while worsening the first one. Thus, FS alone is not appropriate for a PicoDBMS.

4.2 Domain storage

Based on the critique of FS, it follows that a PicoDBMS storage model should guarantee both data and index compactness. Let us first deal with data compactness. Since locality is no longer an issue in our context, pointer-based storage models inspired by MMDBMS [3, 27, 31] can help reducing the data storage cost. The basic idea is to preclude any duplicate value from occuring. This can be achieved by grouping values in domains (sets of unique values). We call this model *domain storage (DS)*. As shown in Fig. 1, tuples reference their attribute values by means of pointers. Furthermore, a domain can be shared among several attributes. This is particularly efficient for enumerated types, which vary on a small and determined set of values[4].

One may wonder about the cost of tuple creation, update, and deletion since they may generate insertion and deletion of values in domains. While these actions are more complex than their FS counterpart, their implementation remains more efficient in the smartcard context, simply because the amount of data to be written is much smaller. To amortize the slight overhead of domain storage, we only store by domain all large attributes (i.e., greater than a pointer size) containing duplicates. Obviously, attributes with no duplicates (e.g., keys) need not be stored by domain but with FS. Variable-size attributes – generally larger than a pointer – can also be advantageously stored in domains even if they do not contain duplicates. The benefit is not storage savings but memory management simplicity (all tuples of all relations become fixed-size) and log compactness (see Sect. 6).

[4] Compression techniques can be advantageously used in conjunction with DS to increase compactness [17].

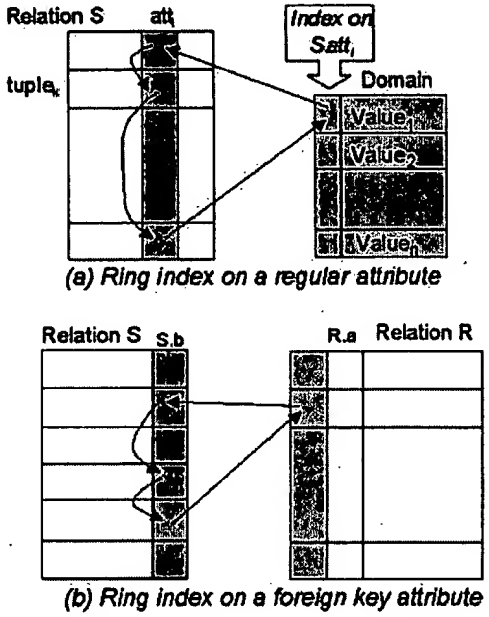


Fig. 2. Ring storage

4.3 Ring storage

We now address index compactness along with data compactness. Unlike disk-based DBMS that favor indices which preserve access locality, smartcards should make intensive use of secondary (i.e., pointer-based) indices. The issue here is to make these indices as compact as possible. Let us first consider select indices. A select index is typically made of two parts: a collection of values and a collection of pointers linking each value to all tuples sharing it. Assuming the indexed attribute varies on a domain, the index's collection of values can be saved since it exactly corresponds to the domain extension. The extra cost incurred by the index is then reduced to the pointers linking index values to tuples.

Let us go one step further and get these pointers almost for free. The idea is to store these *value-to-tuple* pointers in place of the *tuple-to-value* pointers within the tuples (i.e., pointers stored in the tuples to reference their attribute values in the domains). This yields to an index structure which makes a ring from the domain values to the tuples. Hence, we call it *ring index* (see Fig. 2a). However, the ring index can also be used to access the domain values from the tuples and thus serve as data storage model. Thus we call *ring storage (RS)* the storage of a domain-based attribute indexed by a ring. The index storage cost is reduced to its lowest bound, that is, one pointer per domain value, whatever the cardinality of the indexed relation. This important storage saving is obtained at the price of extra work for projecting a tuple to the corresponding attribute since retrieving the value of a ring stored attribute means traversing

on average half of the ring (i.e., up to reaching the domain value).

Join indices [40] can be treated in a similar way. A join predicate of the form ($R.a = S.b$) assumes that $R.a$ and $S.b$ vary on the same domain. Storing both $R.a$ and $S.b$ by means of rings leads to defining a join index. In this way, each domain value is linked by two separate rings to all tuples from R and S sharing the same join attribute value. However, most joins are performed on key attributes, $R.a$ being a primary key and $S.b$ being the foreign key referencing $R.a$. In our model, key attributes are not stored by domain but with FS. Nevertheless, since $R.a$ is the primary key of R, its extension forms precisely a domain, even if not stored outside of R. Since attributes $S.b$ take their values in $R.a$'s domain, they reference $R.a$ values by means of pointers. Thus, the domain-based storage model naturally implements for free a *unidirectional join index* from $S.b$ to $R.a$ (i.e., each S tuple is linked by a pointer to each R tuple matching with it). If traversals from $R.a$ to $S.b$ need to be optimized too, a *bi-directional join index* is required. This can be simply achieved by defining a ring index on $S.b$. Figure 2b shows the resulting situation where each R tuple is linked by a ring to all S tuples matching with it and vice versa. The cost of a bi-directional join index is restricted to a single pointer per R tuple, whatever the cardinality of S. Note that this situation resembles the well-known Codasyl model.

4.4 Storage cost evaluation

Our storage model combines FS, DS, and RS. Thus, the issue is to determine the best storage for each attribute. If the attributes need not be indexed, the choice is obviously between FS and DS. Otherwise, the choice is between RS and FS with a traditional index. Thus, we compare the storage cost for a single attribute, indexed or not, for each alternative. We introduce the following parameters:

- *CardRel*: cardinality of the relation holding the attribute.
- a: average length of the attribute (expressed in bytes).
- p: pointer size (3 bytes will be required to address "large" memory of future cards).
- S: selectivity factor of the attribute. S = *CardDom*/*CardRel*, where *CardDom* is the cardinality of the attribute domain extension (in all models). S measures the redundancy of the attribute (i.e., the same attribute value appears in $1/S$ tuples).

```
Cost(FS) =          CardRel*a          // attribute storage cost in
                                       // the relation
Cost(DS) =          CardRel*p          // attribute storage cost in
                                       // the relation
                  + S*CardRel*a        // values storage cost in
                                       // the domain

Cost(Indexed_FS) = Cost(FS)            // flat attribute storage cost
                  + S*CardRel*a        // value storage cost in the
                                       // index
                  + CardRel*p          // pointer storage cost in
                                       // the index

Cost(RS) =          Cost(DS)           // domain-based attribute
                                       // storage cost
                  + S*CardRel*p        // pointer storage cost in
                                       // the index
```

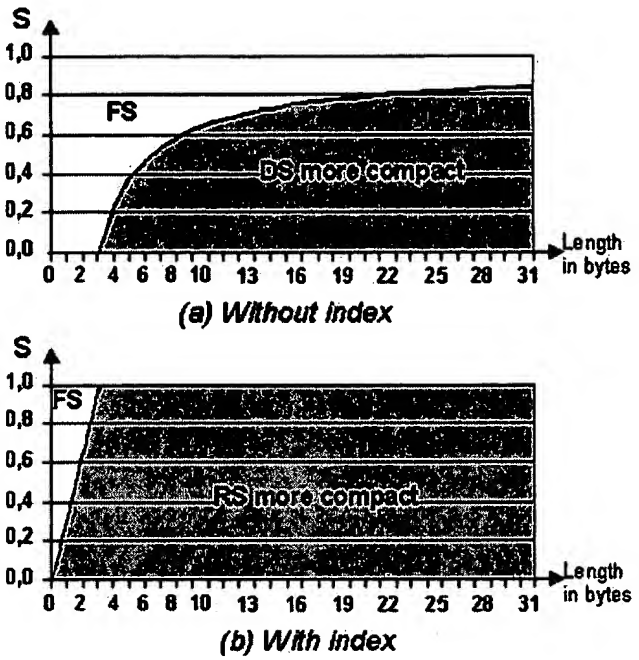


Fig. 3. Storage models' tradeoff

The cost equality between FS and DS gives: $S = (a-p)/a$. The cost equality between Indexed_FS and RS gives:

$$S = a/p$$

Figure 3a shows the different values of S and a for which FS and DS are equivalent. Thus, each curve divides the plan into a gain area for FS (above the curve) and a gain area for DS (under the curve). For values of a less than 3 (i.e., the size of a pointer), FS is obviously always more compact than DS. For higher values of a, DS becomes rapidly more compact than FS except for high values of S. For instance, considering $S = 0.5$, that is the same value is shared by only two tuples, DS outperforms FS for all a larger than 6 bytes. The higher a and the lower S, the better DS. The benefit of DS is thus particularly important for enumerated type attributes. Figure 3b compares Indexed_FS with RS. The superiority of RS is obvious, except for 1- and 2-byte-long key attributes. Thus, Figs. 3a and 3b are guidelines for the database designer to decide how to store each attribute, by considering its size and selectivity.

5 Query processing

Traditional query processing strives to exploit large main memory for storing temporary data structures (e.g., hash tables) and intermediate results. When main memory is not large enough to hold some data, state-of-the-art algorithms (e.g., hybrid hash join [33]) resort to materialization on disk to avoid memory overflow. These algorithms cannot be used for a PicoDBMS because:

- Given the write rule and the lifetime of stable memory, writes in stable memory are proscribed, even for temporary materialization.

- Dedicating a specific RAM area does not help since we cannot estimate its size a priori. Making it small increases the risk of memory overflow, thereby leading to writes in stable memory. Making it large reduces the stable memory area, already limited in a smartcard (RAM rule). Moreover, even a large RAM area cannot guarantee that query execution will not produce memory overflow [9].
- State-of-the-art algorithms are quite sophisticated, which precludes their implementation in a PicoDBMS whose code must be simple, compact, and secure (compactness and security rules).

To solve this problem, we propose query processing techniques that do not use any working RAM area nor incur any writes in stable memory. In the following, we describe these techniques for simple and complex queries, including aggregation and remove duplicates. We show the effectiveness of our solution through a performance analysis.

5.1 Basic query execution without RAM

We consider the execution of *SPJ* (*Select/Project/Join*) queries. Query processing is classically done in two steps. The query optimizer first generates an "optimal" *query execution plan* (*QEP*). The QEP is then executed by the query engine which implements an *execution model* and uses a library of relational operators [17]. The optimizer can consider different shapes of QEP: *left-deep*, *right-deep* or *bushy trees* (see Fig. 4). In a left-deep tree, operators are executed sequentially and each intermediate result is materialized. On the contrary, right-deep trees execute operators in a pipeline fashion, thus avoiding intermediate result materialization. However, they require materializing in memory all left relations. Bushy trees offer opportunities to deal with the size of intermediate results and memory consumption [38].

In a PicoDBMS, the query optimizer should not consider any of these execution trees as they incur materialization. The solution is to only use pipelining with *extreme right-deep trees* where all the operators (including select) are pipelined. As left operands are always base relations, they are already materialized in stable memory, thus allowing us to execute a plan with no RAM consumption. Pipeline execution can be easily achieved using the well-known *Iterator Model* [17]. In this model, each operator is an *iterator* that supports three procedure calls: *open* to prepare an operator for producing an item, *next* to produce an item, and *close* to perform final clean-up. A *QEP* is activated starting at the root of the operator tree and progressing towards the leaves. The dataflow in the model is demand-driven: a child operator passes a tuple to its parent node in response to a *next* call from the parent.

Let us now detail how select, project, and join are performed. These operators can be executed either sequentially or with a ring index. Given the access rule, the use of indices seems always to be the right choice. However, extreme right-deep trees allow us to speed-up a single select on the first base relation (e.g., *Drug.type* in our example), but using a ring index on the other selected attributes (e.g., *Visit.date*) may slow down execution as the rings need to be traversed to retrieve their value. Project operators are pushed up to the tree since no materialization occurs. Note that the final project incurs

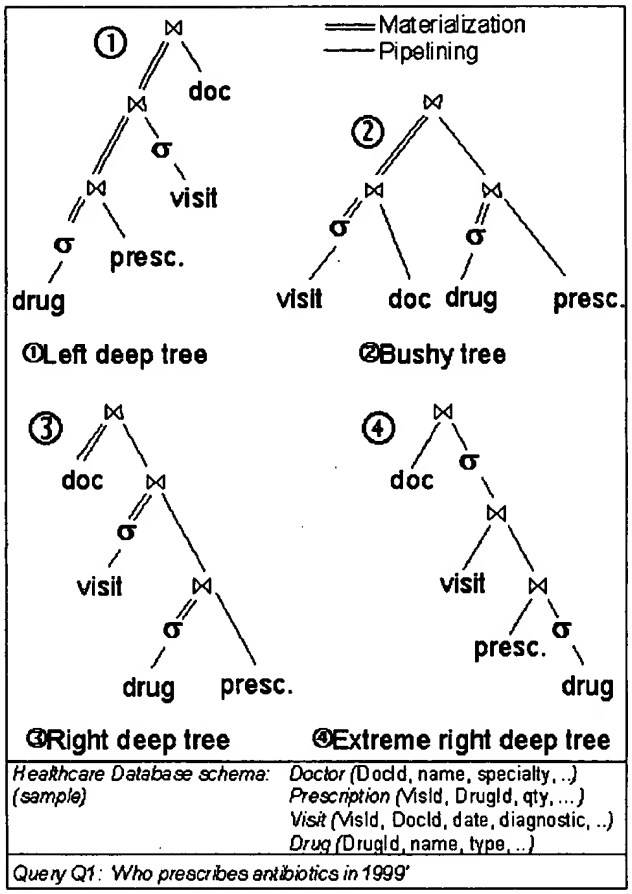


Fig. 4. Several execution trees for query Q1

an additional cost in case of ring attributes. Without indices, joining relations is done by a nested-loop algorithm since no other join technique can be applied without ad hoc structures (e.g., hash tables) and/or working area (e.g., sorting). The cost of indexed joins depends on the way indices are traversed. Consider the indexed join between *Doctor* ($n$tuples) and *Visit* (m tuples) on their key attribute. Assuming a unidirectional index, the join cost is proportional to $n * m$ starting with *Doctor* and to m starting with *Visit*. Assuming now a bi-directional index, the join cost becomes proportional to $n + m$ starting with *Doctor* and to $m^2/2n$ starting with *Visit* (retrieving the doctor associated to each visit incurs traversing half of a ring in average). In the latter case, a naïve nested loop join can be more efficient if the ring cardinality is greater than the target relation cardinality (i.e., when $m > n^2$). In that case, the database designer must clearly choose a unidirectional index between the two relations.

5.2 Complex query execution without RAM

We now consider the execution of aggregate, sort, and duplicate removal operators. At first glance, pipeline execution is not compatible with these operators which are classically performed on materialized intermediate results. Such materialization cannot occur either in the smartcard due to the RAM rule or in the terminal due to the security rule. Note that sorting can be done in the terminal since the output order of the

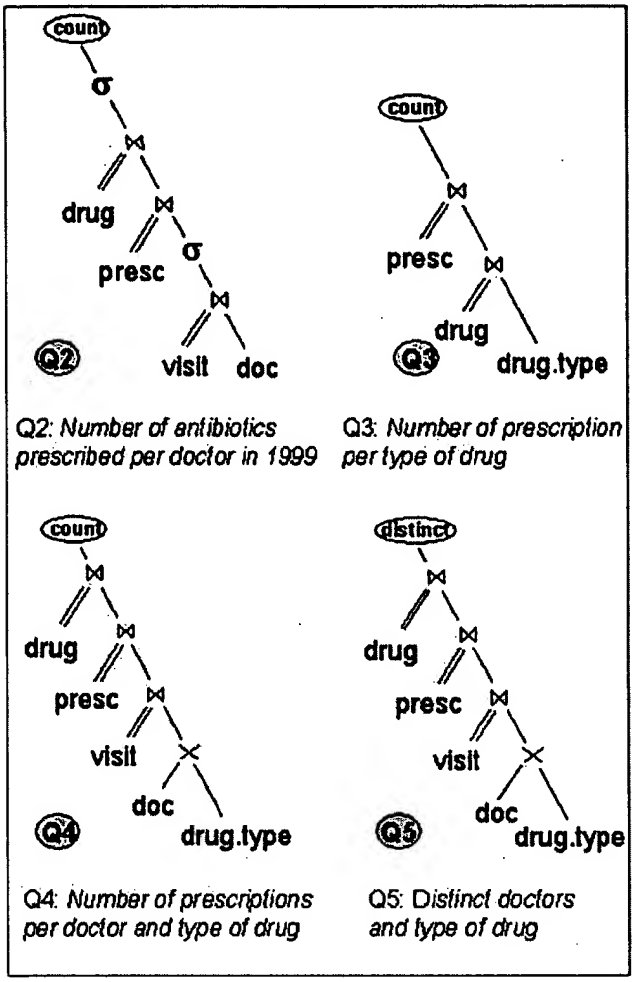


Fig. 5. Four 'complex' query execution plans

result tuples is not significant, i.e., depends on the DBMS algorithms.

We propose a solution to the above problem by exploiting two properties: (i) aggregate and duplicate removal can be done in pipeline if the incoming tuples are still grouped by distinct values; and (ii) pipeline operators are order-preserving since they consume (and produce) tuples in the arrival order. Thus, enforcing an adequate consumption order at the leaf of the execution tree allows pipelined aggregation and duplicate removal. For instance, the extreme right-deep tree of Fig. 4 delivers the tuples naturally grouped by *Drug.id*, thus allowing group queries on that attribute.

Let us now consider query Q2 of Fig. 5. As pictured, executing Q2 in pipeline requires rearranging the execution tree so that relation *Doctor* is explored first. Since *Doctor* contains distinct doctors, the tuples arriving to the *count* operator are naturally grouped by doctors.

The case of Q3 is harder. As the data must be grouped by *type of drugs* rather than by *Drug.id*, an additional join is required between relation *Drug* and domain *drug.type*. Domain values being unique, this join produces the tuples in the adequate order. If domain *Drug.type* does not exist, an operator must be introduced to sort relation *Drug* in pipeline. This can be done by performing n passes on *Drug* where n is the number of distinct values of *Drug.type*.

The case of Q4 is even trickier. The result must be grouped on two attributes (*Doctor.id* and *Drug.type*), introducing the need to start the tree with both relations! The solution is to insert a Cartesian product operator at the leaf of the tree in order to produce tuples ordered by *Doctor.id* and *Drug.type*. In this particular case, the query response time should be approximately n times greater than the same query without the 'group by' clause, where n is the number of distinct *types of drugs*.

Q5 retrieves the distinct couples of *doctor* and *type of prescribed drugs*. This query can be made similar to Q4 by expressing the distinct clause as an aggregate without function (i.e., the query "*select distinct* $a_1, \ldots, a_n$ *from* ... " is equivalent to "*select* $a_1, \ldots, a_n$ *from* ... *group by* $a_1, \ldots, a_n$"). The unique difference is that the computation for a given group, i.e., (*distinct result tuple*) can stop as soon as one tuple has been produced.

5.3 Query optimization

Heuristic optimization is attractive. However, well-known heuristics such as processing select and project first do not work here. Using extreme right-deep trees makes the former impractical and invalidates the latter. Heuristics for join ordering are even more risky considering our data structures. Conversely, there are many arguments for an exhaustive search of the best plan. First, the search space is limited since: (i) there is a single algorithm for each operator, depending on the existing indices; (ii) only extreme right-deep trees are considered; and (iii) typical queries will not involve many relations. Second, exhaustive search using depth-first algorithms do not consume any RAM. Finally, exhaustive algorithms are simple and compact (even if they iterate a lot). Under the assumption that query optimization is required in a PicoDBMS, the remarks above strongly argue in favor of an exhaustive search strategy.

5.4 Performance evaluation

Our proposed query engine can handle fairly complex queries, taking advantage of the read and access rules[5] while satisfying the compactness, write, RAM, and security rules. We now evaluate whether the PicoDBMS performance matches the smartcard application's requirements, that is, any query issued by the application can be performed in reasonable time (i.e., may not exceed the user's patience). Since the PicoDBMS code's simplicity is an important consideration to conform to the compactness and security rules, we must also evaluate which acceleration techniques (i.e., ring indices, query optimization) are really mandatory. For instance, an accelerator reducing the response time from 10 ms to 1 ms is useless in the smartcard context[6]. Thus, unlike traditional performance evaluation, our major concern is on absolute rather than relative performance.

[5] With traditional DBMS, such techniques will induce so many disk accesses that the system would thrash!

[6] With traditional DBMS, such acceleration can improve the transactional throughput.

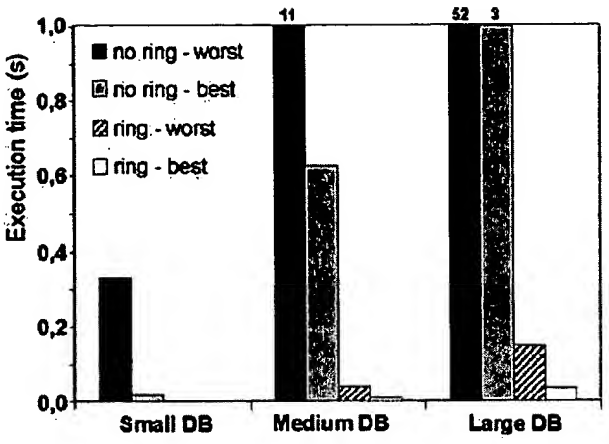


Fig. 6. Performance results for Q1

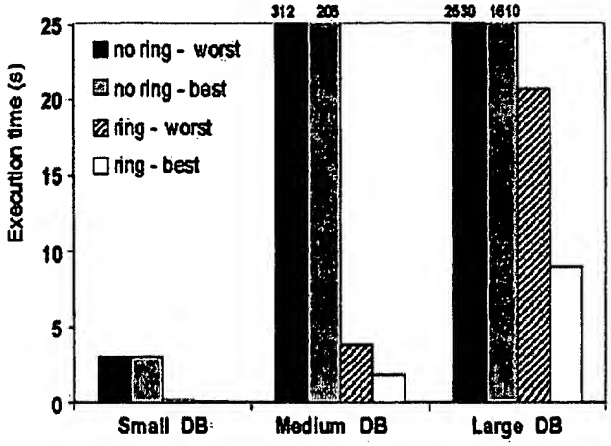


Fig. 7. Performance results for Q4

Evaluating absolute response time is complex in the smartcard environment because all platform parameters (e.g., processor speed, caching strategy, RAM, and EEPROM speed) strongly impact on the measurements[7]. Measuring the performance of our PicoDBMS on Bull's smartcard technology is attractive but introduces two problems. First, Bull's smartcards compatible with database applications are still prototypes [39]. Second, we are interested in providing the most general conclusions (i.e., as independent as possible of smartcard architectures). Therefore, we prefer to measure our query engine on two oldfashioned computers (a PC 486/25 Mhz and a Sun SparcStation 1+) which we felt roughly similar to forthcoming smartcard architectures. For each computer, we vary the system parameters (clock frequency, cache) and perform the experimentation tests. The performance ratios between all configurations were roughly constant (i.e., whatever the query), the slowest configuration (Intel 486 with no cache) performing eight times worse than the fastest (RISC with cache). In the following, we present response times for the slowest architecture to check the viability of our solutions in the worst environment.

We generated three instances of a simplified healthcare database: the *small, medium, and large* databases containing, respectively, (10, 30, 50) doctors, (100, 500, 1,000) visits, (300, 2,000, 5,000) prescriptions, and (40, 120, 200) drugs. Although we tested several queries, we describe below only the two most significant. Query Q1, which contains three joins and two selects on *Visit* and *Drug* (with selectivities of 20% and 5%), is representative of medium-complexity queries. Query Q4, which performs an aggregate on two attributes and requires the introduction of a Cartesian product, is representative of complex queries. For each query, we measure the performance for all possible query execution plans, excluding those which induce additional Cartesian product, varying the storage choices (with and without select and join ring indices). Figures 6 and 7 show the results for both best and worst plans on databases built with or without join indices.

Considering SPJ queries, the PicoDBMS performance clearly matches the application's requirements as soon as join rings are used. Indeed, the performance with join rings is at most 146 ms for the largest database and with the worst execution plan. With small databases, all the acceleration techniques can be discarded, while with larger ones, join rings remain necessary to obtain good response time. In that case, the absolute gain (110 ms) between the best and the worst plan does not justify the use of a query optimizer.

The performance of aggregate queries is clearly the worst because they introduce a Cartesian product at the leaf of the execution tree. Join rings are useful for medium and large databases. With large databases, the optimizer turns out to be necessary since the worst execution plan with join rings achieves a rather long response time (20.6 s).

The influence of ring indices for selects (not shown) is insignificant. Depending on the selectivity, it can bring slight improvement or overhead on the results. Although it may achieve an important relative speed-up for the select itself, the absolute gain is not significant considering the small influence of select on the global query execution cost (which is not the case in disk-based DBMS). Select ring indices are, however, useful for queries with aggregates or duplicate removal, that can result in a join between a relation and the domain attribute. In that case, the select index plays the role of a join index, thereby generating a significant gain on large relations and large domains.

Thus, this performance evaluation shows that our approach is feasible and that join indices are mandatory in all cases while query optimization turns out to be useful only with large databases and complex queries.

6 Transaction management

Like any data server, a PicoDBMS must enforce the well-known transactional ACID properties [8] to guarantee the consistency of the local data it manages as well as be able to participate in distributed transactions. We discuss below these properties with respect to a PicoDBMS.

- *Atomicity*: *local atomicity* means that the set of actions performed by the PicoDBMS on a transaction's behalf is made persistent following the *all or nothing* scheme. *Global atomicity*: this means that all data servers – including the PicoDBMS – accessed by a distributed transaction

[7] With traditional DBMS, very slow disk access allows us to ignore finer parameters.

agree on the same transaction outcome (either commit or rollback). The distinguishing features of a PicoDBMS regarding atomicity are no demarcation between main memory and persistent storage, the dramatic cost of writes, and the fact that they cannot be deferred.

- *Consistency*: this property ensures that the actions performed by the PicoDBMS satisfy all integrity constraints defined on the local data. Considering that traditional integrity constraint management can be used, we do not discuss it any further.
- *Isolation*: this property guarantees the serializability of concurrent executions. A PicoDBMS manages personal data and is typically single-user[8]. Furthermore, smartcard operating systems do not even support multithreading. Therefore, isolation is useless here.
- *Durability*: durability means that committed updates are never lost whatever the situation (i.e., even in case of a media failure). Durability cannot be enforced locally by the PicoDBMS because the smartcard is more likely to be stolen, lost or destroyed than a traditional computer. Indeed, mobility and smallness play against safety. Consequently, durability must be enforced through the network. The major issue is then preserving the privacy of data while delegating the durability to an external agent.

The remainder of this section addresses local atomicity, global atomicity, and durability.

6.1 Local atomicity

There are basically two ways to perform updates in a DBMS. The updates are either performed on *shadow objects* that are atomically integrated in the database at commit time or done *in place* (i.e., the transaction updates the shared copy of the database objects) [8]. We discuss these two traditional models below.

- *Shadow update*: This model is rarely employed in disk-based DBMSs because it destroys data locality on disk and increases concurrent updates on the catalog. In a PicoDBMS, disk locality and concurrency are not a concern. This model has been shown to be convenient for smartcards equipped with a small Flash memory [25]. However, it is poorly adapted to pointer-based storage models like RS since the object location changes at every update. In addition, the cost incurred by shadowing grows with the memory size. Indeed, either the granularity of the shadow objects increases or the paths to be duplicated in the catalog become longer. In both cases, the writing cost – which is the dominant factor – increases.
- *Update in-place*: write-ahead logging (*WAL*) [8] is required in this model to undo the effects of an aborted transaction. Unfortunately, the relative cost of WAL is much higher in a PicoDBMS than in a traditional disk-based DBMS which uses buffering to minimize I/Os. In a smartcard, the log must be written for each update since each update becomes immediately persistent. This roughly doubles the cost of writing.

[8] Even if the data managed by the PicoDBMS are shared among multiple users (e.g., as in the healthcare application), the PicoDBMS serves a single user at a time.

Despite its drawbacks, *update in-place* is better suited than *shadow update* for a PicoDBMS because it accommodates pointer-based storage models and its cost is insensitive to the rapid growth of stable memory capacity. We also propose two optimizations to *update in-place*:

- *Pointer-based logging*: traditional WAL logs the values of all modified data. RS allows a finer granularity by logging pointers in place of values. The smallest the log records, the cheapest the WAL. The logging process must consider two types of information:
- *Values*: in case of a tuple update, the log record must contain the tuple address and the old attribute values, that is a pointer for all RS stored attributes and a regular value for FS stored attributes. In case of a tuple insertion or deletion, assuming each tuple header contains a status bit (i.e., dead or alive), only the tuple address has to be logged in order to recover its state.
- *Rings*: tuple insertion, deletion, and update (of a ring attribute) modify the structure of each ring traversing the corresponding tuple t. Since a ring is a circular chain of pointers, recovering its state means recovering the *next* pointer of t's predecessor (let us call it t_{pred}). The information to restore in $t_{pred}.next$ is either t's address if t has been updated or deleted, or $t.next$ if t has been inserted. t's address already belongs to the log (see above) and $t.next$ does not have to be logged since t's content still exists in stable storage at recovery time. The issue is how to identify t_{pred} at recovery time. Logging this information can be saved at the price of traversing the whole ring starting from t, until reaching t again. Thus, ring recovery comes for free in terms of logging.
- *Garbage-collecting values*: insertion and deletion of domain values (domain values are never modified) should be logged as any other updates. This overhead can be avoided by implementing a deferred garbage collector that destroys all domain values no longer referenced by any tuple. Garbage-collecting a domain amounts to execute an ad hoc semi-join operator between the domain and all relations varying on it which discards the domain values that do not match[9]. The benefit of this solution is threefold: (i) the lazy deletion of unreferenced values does not entail the storage model coherency; (ii) garbage-collecting domain values is required anyway by RS (even in the absence of transaction control); and (iii) a deferred garbage-collector can be implemented without reference counters, thereby saving storage space. The deferred garbage collector cannot work in the background since smartcards do not yet support multi-threading. The most pragmatic solution is to launch it manually when the card is nearly full. An alternative to this manual procedure is to execute the garbage collector automatically at each card connection on a very small subset of the database (so that its cost remains hidden to the user). Garbage-collecting the database in such an incremental way is straightforward since domain values are examined one after the other.

[9] Unlike reachability algorithms that start from the persistent roots and need marking [6], the proposed garbage-collector starts from the persistent leaves (i.e., the domain values) and exploits them one after the other, in a pipelined fashion (thus, it conforms to the RAM rule).

The update in-place model along with pointer-based logging and deferred garbage-collector reduces logging cost to its lowest bound, that is, a tuple address for inserted and deleted tuples, and the values of updated attributes (again, a pointer for DS and RS stored attributes).

6.2 Global atomicity

Global atomicity is traditionally enforced by an *atomic commitment protocol* (*ACP*). The most well known and widely used ACP is 2PC [8]. While extensively studied [19] and standardized [21, 29, 41], 2PC suffers from the following weaknesses in our context:

- *Need for a standard prepared state*: any server must externalize the standard *Xa* interface [41] to participate to 2PC. Unfortunately, ISO defines a transactional interface for smartcards but it does not cover distributed transactions [24]. In addition, participating to 2PC requires building a local prepared state that consumes valuable resources.
- *Disconnection means aborting*: a smartcard can be extracted from its terminal or its mobile host (e.g., a cellular phone) can be temporarily unreachable during 2PC. A participant's disconnection leads 2PC to abort the transaction even if all its operations have been successfully executed.
- *Badly adapted to moving participants*: the 2PC incurs two message rounds to commit a transaction. Considering the high cost of wireless communication, the overhead is significant for mobile terminals equipped with a smartcard reader (e.g., PDA, cellular phones).

As its name indicates, 2PC has two phases: the *voting* phase and the *decision* phase. The voting phase is the means by which the coordinator checks whether or not the participants can locally guarantee the ACID properties of the distributed transaction. The decision is *commit* if all participants vote *yes* and *abort* otherwise. Thus, the voting phase introduces an uncertainty period at transaction termination that leads to the aforementioned drawbacks.

Variations of *one-phase commit* protocols (*1PC*) have been recently proposed [2, 4, 35]. As stated in [2], 1PC eliminates the voting phase of 2PC by enforcing the following properties on the participant's behavior: (1) all operations are acknowledged before the 1PC is launched; (2) there are no deferred integrity constraints; (3) all participants are ruled by a rigorous concurrency control scheduler; and (4) all updates are logged on stable storage before 1PC is launched. These assumptions guarantee, respectively, the A, C, I, D properties before the ACP is launched. Then, the ACP reduces to a single phase, that is broadcasting the coordinator's decision to all participants (this decision is commit if all transaction's operations have been successfully executed and abort otherwise). If a crash or a disconnection precludes a participant from conforming to this decision, the corresponding transaction branch is simply forward recovered (potentially at the next reconnection). While the assumptions on the participant's behavior seem constraining in the general case, they are quite acceptable in the smartcard context [10]. Property (1) is common to all ACPs and is enforced by the ISO7816 standard [22]; property (2) conforms to the fact that PicoDBMS have lighter capabilities than full-fledged DBMS; and property (3) is satisfied by definition since smartcards do not support parallel executions. Property (4) is discussed in Sect. 6.3.

Eliminating the voting phase of the ACP solves altogether the three aforementioned problems. However, one may wonder about the interoperability between transaction managers and data managers supporting different protocols (either 1PC or 2PC). We have shown in [1] that the participation of legacy (i.e., 2PC compliant) data managers in 1PC is straightforward. Conversely, the participation of 1PC compliant data managers (e.g., a smartcard) in the 2PC can be achieved by associating a *log agent* to each participant. The role of the log agent is twofold. First, it manages the data manager's part of the 1PC's coordinator log, forces it to stable storage during the 2PC prepare phase, and exploits it if the transaction branch needs to be forward-recovered. Second, it translates the 2PC interface into that of 1PC. The log agent can be located on the terminal, so that the benefit of 1PC is lost for the terminal but it is preserved for the smartcard.

6.3 Durability

Most 1PC protocols assume that the coordinator is in charge of logging all participants' updates before triggering the ACP (all these protocols belong to the coordinator log family). *Coordinator log* [35] and *implicit yes vote* [4] assume that the participants piggyback their log records on the acknowledgment messages of each operation while *coordinator logical log* [2] assumes that the coordinator logs all operations sent to each participant. In all cases, the durability of the distributed transaction relies on the coordinator log. Thus, 1PC is a means by which global atomicity and durability can be solved altogether, at the same price.

Two issues remain to be solved: (i) where to store the coordinator log; and (ii) how to preserve the security rule, that is, how to make the log content as secure as the data stored in the smartcard. Since the log must sustain any kind of failure, it must be stored on the network by a trustee server (e.g., a public organism, a central bank, the card issuer, etc.). If some transactions are executed in disconnected mode (e.g., on a mobile terminal), the durability will be effective only at the time the terminal reconnects to the network. Protecting the log content against attacks imposes encryption. The way encryption is performed depends on the model of logging. If the coordinator log is fed by the log records piggybacked by the participants, the smartcard can encrypt them with an algorithm based on a private key (e.g., DES [28]). Otherwise (i.e., if the *coordinator logical log* scheme is selected), the smartcard can provide the coordinator with a public key that will be used by the coordinator itself to encrypt its log [32].

6.4 Transaction cost evaluation

The goal of this section is to approximate the time required by a representative update transaction. The objective is to confirm whether or not the write performance of smartcards assumed in this paper is acceptable for database applications like health cards. To this end, we estimate the time required to create a tuple in a relation, including the creation of domain values,

the insertion of the tuple in the rings potentially defined on this relation and the log time. Let us introduce the following parameters, in addition to those already defined in Sect. 4.4:

- *nbAttFS*: number of FS stored attributes
- *nbAttDS*: number of DS stored attributes
- *nbAttRS*: number of RS stored attributes
- *w*: size of a word (4 bytes in a 32-bit card)
- *t*: time to write one word in stable storage (5 ms in the worst case)

```
Cost(insertTuple) =
  ([(nbAttFS*a + nbAttDS*p + nbAttRS*p)/w]   // ①
  + (nbAttRS + nbAttDS) * S * [a/w]          // ②
  + nbAttRS * [p/w]                          // ③
  + [p/w]                                    // ④
  ) * t                                      // ⑤
```

① Tuple size
② Domain values size. S ≈ probability to create a new domain value
③ Ring pointers to be updated
④ Log record size
⑤ Write time

Let us consider a representative transaction executed on the healthcare. This transaction inserts a new tuple in *Doctor* and *Visit* and five tuples in *Prescription* and *Drug*. This is somehow a worst case for this application in the sense that the visited doctor is a new one and prescribes five new drugs. The considered attribute distribution is as follows:

Doctor	*(nbAttFS=3, nbAttDS=4, nbAttRS=0),*
Visit	*(nbAttFS=2, nbAttDS=3, nbAttRS=2),*
Prescription	*(nbAttFS=1, nbAttDS=1, nbAttRS=2),*
Drug	*(nbAttFS=2, nbAttDS=4, nbAttRS=0).*

The average attribute length a is fixed to 10 bytes. Figure 8 plots the update transaction execution time depending on S ($S = 0$ means that all attribute values already exist in the domains, while $S = 1$ means that all these values need be inserted in the domains).

The figure is self-explanatory. Note that the logging cost represents less than 3% of the total cost. This simple analysis shows that the time expected for this kind of transaction (less than 1 s) is clearly compatible with the healthcare application's requirements.

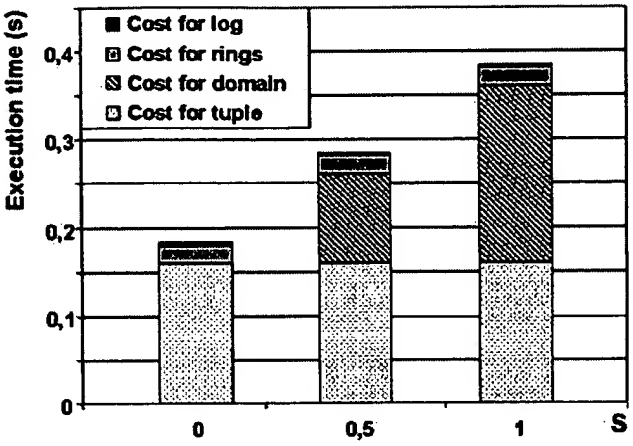


Fig. 8. Performance of a typical update transaction

7 Conclusion

As smartcards become more and more versatile, multi-application, and powerful, the need for database techniques arises. However, smartcards have severe hardware limitations which make traditional database technology irrelevant. The major problem is scaling down database techniques so they perform well under these limitations. In this paper, we addressed this problem and proposed the design of a PicoDBMS, concentrating on the components which require non-traditional techniques (storage manager, query manager, and transaction manager).

This paper makes several contributions. First, we analyzed the requirements for a PicoDBMS based on a healthcare application which is representative of personal folder applications and has strong database requirements. We showed that the minimal functionality should include select/project/join/aggregate, access right management, and views as well as transaction's atomicity and durability.

Second, we gave an in-depth analysis of the problem by considering the smartcard hardware trends. Based on this analysis, we assumed a smartcard with a reasonable stable memory of a few megabytes and a small RAM of some kilobytes, and we derived design rules for a PicoDBMS architecture.

Third, we proposed a new highly compact storage model that combines flat storage (FS), domain storage (DS), and ring storage (RS). Ring storage reduces the indexing cost to its lowest bound. Based on performance evaluation, we derived guidelines to decide the best way to store an attribute.

Fourth, we proposed query processing techniques which handle complex query plans with no RAM consumption. This is achieved by considering extreme right-deep trees which can pipeline all operators of the plan including aggregates. We also argued that, if query optimization is needed, the strategy should be exhaustive search. We measured the performance of our execution model with an implementation of our query engine on two old-fashioned computers which we configured to be similar to forthcoming smartcard architectures. We showed that the resulting performance matches the smartcard application's requirements.

Finally, we proposed techniques for transaction atomicity and durability. Local atomicity is achieved through update in-place with two optimizations which exploit the storage model: pointer-based logging and garbage collection of domain values. Global atomicity and durability are enforced by 1PC which is easily applicable in the smartcard context and more efficient than 2PC. We showed that the performance of typical update transactions is acceptable for representative applications like the health card.

This work is done in the context of a new project with Bull Smart Cards and Terminals. The next step is to port our PicoDBMS prototype on Bull's smartcard new technology, called *OverSoft* [12], and to assess its functionality and performance on real-world applications. To this end, a benchmark dedicated to PicoDBMS must be set up. We also plan to address open issues such as protected logging for durability, query execution on encrypted data (e.g., stored in an external Flash), and statistics maintenance on a population of cards.

References

1. Abdallah M., Bobineau C., Guerraoui R., Pucheral P: Specification of the transaction service. Esprit project OpenDREAMS-II n° 25262, Deliverable n° R13, 1998
2. Abdallah M., Guerraoui R., Pucheral P.: One-phase commit: does it make sense? Int. Conf. on Parallel and Distributed Systems (ICPADS), 1998
3. Ammann A., Hanrahan M., Krishnamurthy R.: design of a memory resident DBMS. IEEE COMPCON, 1985
4. Al-Houmaily Y., Chrysanthis P.K.: Two-phase commit in gigabit-networked distributed databases. Int. Conf. on Parallel and Distributed Computing Systems (PDCS), 1995
5. Anderson R., Kuhn M.: Tamper resistance – a cautionary note. USENIX Workshop on Electronic Commerce, 1996
6. Amsaleg L., Franklin M.J., Gruber O.: Efficient incremental garbage collection for client-server object database systems. Int. Conf. on Very Large Databases (VLDB), 1995
7. Bobineau C., Bouganim L., Pucheral P., Valduriez P.: PicoDBMS: scaling down database techniques for the smartcard (Best Paper Award). Int. Conf. on Very Large Databases (VLDB), 2000
8. Bernstein P.A., Hadzilacos V., Goodman N.: Concurrency control and recovery in database systems. Addison-Wesley, Reading, Mass., USA,1987
9. Bouganim L., Kapitskaia O., Valduriez P.: Memory-adaptive scheduling for large query execution. Int. Conf. on Information and Knowledge Management (CIKM), 1998
10. Bobineau C., Pucheral P., Abdallah M.: A unilateral commit protocol for mobile and disconnected computing. Int. Conf. On Parallel and Distributed Computing Systems (PDCS), 2000
11. van Bommel F.A., Sembritzki J., Buettner H.-G.: Overview on healthcard projects and standards. Health Cards Int. Conf., 1999
12. Bull S.A.: Bull unveils iSimplify! the personal portable portal. Available at: http://www.bull.com:80/bull_news/
13. Carrasco L.C.: RDBMS's for Java cards? What a senseless idea! Available at: www.sqlmachine.com, 1999
14. DataQuest.: Chip card market and technology charge ahead. MSAM-WW-DP-9808, 1998
15. Dipert B.: FRAM: Ready to ditch niche? EDN Access Magazine, Cahners, London, 1997
16. Gemplus.: SIM Cards: From kilobytes to megabytes. Available at: www.gemplus.fr/about/pressroom/, 1999
17. Graefe G.: Query evaluation techniques for large databases. ACM Comput Surv, 25(2), 1993
18. Graefe G.: The new database imperatives. Int. Conf. on Data Engineering (ICDE), 1998
19. Gray J., Reuter A.: Transaction processing. Concepts and Techniques. Morgan Kaufmann, San Francisco, 1993
20. IBM Corporation.: DB2 Everywhere – administration and application programming guide. IBM Software Documentation, SC26-9675-00, 1999
21. International Standardization Organization (ISO).: Information technology - open systems interconnection - distributed transaction processing. ISO/IEC 10026, 1992
22. International Standardization Organization (ISO).: Integrated circuit(s) cards with contacts – part 3: electronic signal and transmission protocols. ISO/IEC 7816-3, 1997
23. International Standardization Organization (ISO).: Integrated circuit(s) cards with contacts – part 1: physical characteristics. ISO/IEC 7816-1, 1998
24. International Standardization Organization (ISO).: Integrated circuit(s) cards with contacts – part 7: interindustry commands for structured card query language (SCQL). ISO/IEC 7816-7, 1999
25. Lecomte S., Trane P.: Failure recovery using action log for smartcards transaction based system. IEEE Online Testing Workshop, 1997
26. Microsoft Corporation.: Windows for smartcards toolkit for visual basic 6.0. Available at: www.microsoft.com/windowsce/smartcard/, 2000
27. Missikov M., Scholl M.: Relational queries in a domain based DBMS. ACM SIGMOD Int. Conf. on Management of Data, 1983
28. National Institute of Standards and Technology.: Announcing the Data Encryption Standard (DES). FIPS PUB 46-2, 1993
29. Object Management Group.: Object transaction service. Document 94.8.4, OMG editor, 1994
30. Oracle Corporation.: Oracle 8i Lite - Oracle Lite SQL reference. Oracle documentation, A73270-01, 1999
31. Pucheral P., Thévenin J.M., Valduriez P.: Efficient main memory data management using the DBGraph storage model. Int. Conf. on Very Large Databases (VLDB), 1990
32. RSA Laboratories.: PKCS # 1: RSA Encryption Standard. RSA Laboratories Technical Note, v.1.5, 1993
33. Schneider D., DeWitt D.: A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. ACM-SIGMOD Int. Conf., 1989
34. Schneier B., Shostack A.: Breaking up is hard to do: modeling security threats for smart cards. USENIX Symposium on Smart Cards, 1999
35. Stamos J., Cristian F.: A low-cost atomic commit protocol. IEEE Symposium on Reliable Distributed Systems, 1990
36. Sun Microsystems.: JavaCard 2.1 application programming interface specification. JavaSoft documentation, 1999
37. Sybase Inc.: Sybase adaptive server anywhere reference. CT75KNA, 1999
38. Shekita E., Young H., Tan K.L.: Multi-join optimization for symmetric multiprocessors. Int. Conf. on Very Large Data Bases (VLDB), 1993
39. Tual J.-P.: MASSC: a generic architecture for multiapplication smart cards. IEEE Micro J, N° 0272-1739/99, 1999
40. Valduriez P.: Join indices. ACM Trans. Database Syst, 12(2), 1987
41. X/Open.: Distributed transaction processing: reference model. X/Open Guide, Version 3. G307., X/Open Company Limited, 1996

Programming Techniques and Data Structures
Ellis Horowitz Editor

Implementations for Coalesced Hashing

Jeffrey Scott Vitter
Brown University

The coalesced hashing method is one of the faster searching methods known today. This paper is a practical study of coalesced hashing for use by those who intend to implement or further study the algorithm. Techniques are developed for tuning an important parameter that relates the sizes of the address region and the cellar in order to optimize the average running times of different implementations. A value for the parameter is reported that works well in most cases. Detailed graphs explain how the parameter can be tuned further to meet specific needs. The resulting tuned algorithm outperforms several well-known methods including standard coalesced hashing, separate (or direct) chaining, linear probing, and double hashing. A variety of related methods are also analyzed including deletion algorithms, a new and improved insertion strategy called varied-insertion, and applications to external searching on secondary storage devices.

CR Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics—*performance measures*; E.2 [**Data**]: Data Storage Representations—*hash-table representations*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sorting and searching*; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: analysis of algorithms, coalesced hashing, hashing, data structures, databases, deletion, asymptotic analysis, average-case, optimization, secondary storage, assembly language

This research was supported in part by a National Science Foundation fellowship and by National Science Foundation grants MCS-77-23738 and MCS-81-05324.

Author's Present Address: Jeffrey Scott Vitter, Department of Computer Science, Box 1910, Brown University, Providence, RI 02912.

1. Introduction

One of the primary uses today for computer technology is information storage and retrieval. Typical searching applications include dictionaries, telephone listings, medical databases, symbol tables for compilers, and storing a company's business records. Each package of information is stored in computer memory as a *record*. We assume there is a special field in each record, called the *key*, that uniquely identifies it. The job of a searching algorithm is to take an input K and return the record (if any) that has K as its key.

Hashing is a widely used searching technique because no matter how many records are stored, the average search times remain *bounded*. The common element of all hashing algorithms is a predefined and quickly computed *hash function*

$$hash: \{\text{all possible keys}\} \rightarrow \{1, 2, \ldots, M\}$$

that assigns each record to a *hash address* in a uniform manner. (The problem of designing hash functions that justify this assumption, even when the distribution of the keys is highly biased, is well-studied [7, 2].) Hashing methods differ from one another by how they resolve a *collision* when the hash address of the record to be inserted is already occupied.

This paper investigates the *coalesced hashing* algorithm, which was first published 22 years ago and is still one of the faster known searching methods [16, 7]. The total number of available storage locations is assumed to be *fixed*. It is also convenient to assume that these locations are contiguous in memory. For the purpose of notation, we shall number the hash table slots $1, 2, \ldots, M'$. The first M slots, which serve as the range of the hash function, constitute the *address region*. The remaining $M'-M$ slots are devoted solely to storing records that collide when inserted; they are called the *cellar*. Once the cellar becomes full, subsequent colliders must be stored in empty slots in the address region and, thus, may trigger more collisions with records inserted later.

For this reason, the search performance of the coalesced hashing algorithm is very sensitive to the relative sizes of the address region and cellar. In Sec. 4, we apply the analytic results derived in [10, 11, 13] in order to optimize the ratio of their sizes, $\beta = M/M'$, which we call the *address factor*. The optimizations are based on two performance measures: the number of probes per search and the running time of assembly language versions. There is no unique best choice for β—the optimum address factor depends on the type of search, the number of inserted records, and the performance measure chosen—but we shall see that the compromise choice $\beta \approx 0.86$ works well in many situations. The method can be further turned to meet specific needs.

Section 5 shows that this tuned method dominates several popular hashing algorithms including standard coalesced hashing (in which $\beta = 1$), separate (or direct)

chaining, linear probing, and double hashing. The last three sections deal with variations and different implementations for coalesced hashing including deletion algorithms, alternative insertion methods, and external searching on secondary storage devices.

This paper is designed to provide a comprehensive treatment of the many practical issues concerned with the implementation of the coalesced hashing method. Readers interested in the theoretical justification of the results in this paper can consult [10, 11, 13, 14, 1].

2. The Coalesced Hashing Algorithm

The algorithm works like this: Given a record with key K, the algorithm searches for it in the hash table, starting at location $hash(K)$ and following the links in the chain. If the record is present in the table, then it is found and the search is *successful*; otherwise, the end of the chain is reached and the search is *unsuccessful*. For simplicity, we assume that the record is inserted whenever the search ends unsuccessfully, according to the following rule: If position $hash(K)$ is empty, then the record is stored at that location; else, it is placed in the largest-numbered empty slot in the table and is linked to the end of the chain. This has the effect of putting the first $M'-M$ colliders into the cellar.

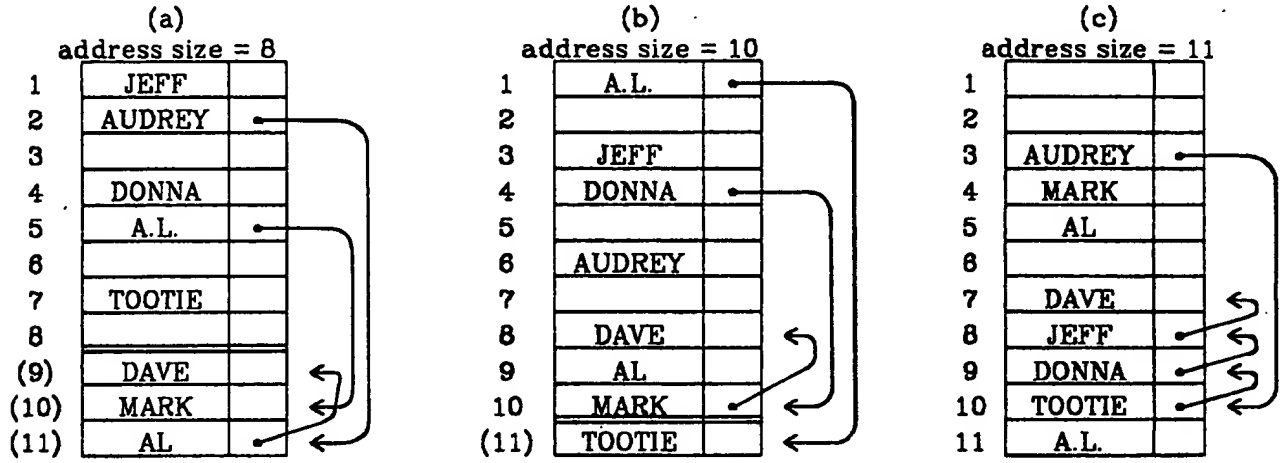
Coalesced hashing is a generalization of the well-known separate (or direct) chaining method. The separate chaining method halts with overflow when there is no more room in the cellar to store a collider. The example in Fig. 1(a) can be considered to be an example of both coalesced hashing and separate chaining, because the cellar is large enough to store the three colliders.

Figures 1(b) and 1(c) show how the two methods differ. The cellar contains only one slot in the example in Fig. 1(b). When the key MARK collides with DONNA at slot 4, the cellar is already full. Separate chaining would report overflow at this point. The coalesced hashing method, however, stores the key MARK in the largest-numbered empty space (which is location 10 in the address region). This causes a later collision when DAVE hashes to position 10, so DAVE is placed in slot 8 at the end of the chain containing DONNA and MARK. The method derives its name from this "coalescing" of records with different hash addresses into single chains.

The average number of probes per search shows marked improvement in Fig. 1(b), even though coalescing has occurred. Intuitively, the larger address region spreads out the records more evenly and causes fewer collisions, i.e., the hash function can be thought of as "shooting" at a bigger target. The cellar is now too small to store these fewer colliders, so it overflows. Fortunately, this overflow occurs late in the game, and the pileup phenomenon of coalescing is not significant enough to counteract the benefits of a larger address region. However, in the extreme case when $M = M' = 11$ and there is no cellar (which we call *standard coalesced hashing*), coalescing begins too early and search time worsens (as typified by Figure 1(c)). Determining the optimum address factor $\beta = M/M'$ is a major focus of this paper.

The first order of business before we can start a detailed study of the coalesced hashing method is to formalize the algorithm and to define reasonable measures of search performance. Let us assume that each

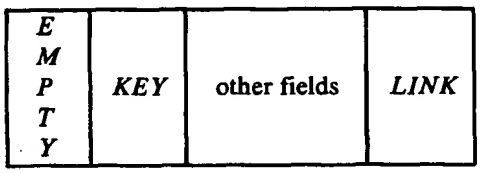
Fig. 1. Coalesced hashing, $M' = 11$, $N = 8$. The sizes of the address region are (a) $M = 8$, (b) $M = 10$, and (c) $M = 11$.



Keys:		A.L.	AUDREY	AL	TOOTIE	DONNA	MARK	JEFF	DAVE
Hash Addresses:	(a)	5	2	2	7	4	5	1	2
	(b)	1	6	9	1	4	4	3	10
	(c)	11	3	5	3	10	4	10	9

average # probes per successful search: (a) 12/8 = 1.5, (b) 11/8 = 1.375, (c) 14/8 = 1.75.

of the M' contiguous slots in the coalesced hash table has the following organization:



For each value of i between 1 and M', $EMPTY[i]$ is a one-bit field that denotes whether the ith slot is unused, $KEY[i]$ stores the key (if any), and $LINK[i]$ is either the index to the next spot in the chain or else the null value 0.

The algorithms in this article are written in the English-like style used by Knuth in order to make them readily understandable to all and to facilitate comparisons with the algorithms contained in [7, 4, 12]. Block-structured languages, like PL/I and Pascal, are good for expressing complicated program modules; however, they are not used here, because hashing algorithms are so short that there is no reason to discriminate against those who are not comfortable with such languages.

Algorithm C (*Coalesced hashing search and insertion*). This algorithm searches an M'-slot hash table, looking for a given key K. If the search is unsuccessful and the table is not full, then K is inserted.

The size of the address region is M; the hash function *hash* returns a value between 1 and M (inclusive). For convenience, we make use of slot 0, which is always empty. The global variable R is used to find an empty space whenever a collision must be stored in the table. Initially, the table is empty, and we have $R = M' + 1$; when an empty space is requested, R is decremented until one is found. We assume that the following initializations have been made before any searches or insertions are performed: $M \leftarrow \lceil \beta M' \rceil$, for some constant $0 < \beta \leq 1$; $EMPTY[i] \leftarrow$ **true**, for all $0 \leq i \leq M'$; and $R \leftarrow M' + 1$.

C1. [Hash.] Set $i \leftarrow hash(K)$. (Now $1 \leq i \leq M$.)

C2. [Is there a chain?] If $EMPTY[i]$, then go to step C6. (Otherwise, the ith slot is occupied, so we will look at the chain of records that starts there.)

C3. [Compare.] If $K = KEY[i]$, the algorithm terminates *successfully*.

C4. [Advance to next record.] If $LINK[i] \neq 0$, then set $i \leftarrow LINK[i]$ and go back to step C3.

C5. [Find empty slot.] (The search for K in the chain was unsuccessful, so we will try to find an empty table slot to store K.) Decrease R one or more times until $EMPTY[R]$ becomes **true**. If $R = 0$, then there are no more empty slots, and the algorithm terminates *with overflow*. Otherwise, append the Rth cell to the chain by setting $LINK[i] \leftarrow R$; then set $i \leftarrow R$.

C6. [Insert new record.] Set $EMPTY[i] \leftarrow$ **false**, $KEY[i] \leftarrow K$, $LINK[i] \leftarrow 0$, and initialize the other fields in the record. ■

In this paper, we concern ourselves with measuring the *searching phase* of Algorithm C and ignore for the most part the insertion time in steps C5 and C6. (The time for step C5 is not significant, because the total number of times R is decremented over the course of all the insertions cannot be more than the number of inserted records; hence, the amortized expected number of decrements is at most 1. The decrementing operation can also be done in parallel with steps C1–C4.) Our primary measure of search performance is the *number of probes per search*, which is the number of different table slots that are accessed while searching. In Algorithm C, this quantity is equal to

$$\max\{1, \text{number of times step C3 is performed}\}$$

For example, in Fig. 1(b), the unsuccessful searches for keys A.L. and TOOTIE (immediately prior to their insertions) each took one probe, while a successful search for DAVE would take two probes.

The average performance of the algorithm is obtained by assuming that all searches and insertions are *random*. The Appendix contains a discussion of the probability model as well as the formulas for the expected number of probes in unsuccessful and successful searches.

3. Assembly Language Implementation

Even though probe-counting gives us a good idea of search performance, other factors (such as the complexity of the search loop and the overhead is computing the hash address) also affect the running time when Algorithm C is programmed for a real computer. For completeness, we optimize the running time of assembly language versions of coalesced hashing.

We choose to program in assembly language rather than in some high-level language like Fortran, PL/I, or Pascal, in order to achieve *maximum possible efficiency*. Top efficiency is important in large-scale applications of hashing, but it can also be achieved in smaller systems with little extra effort, because hashing algorithms are so short that implementing them (even in assembly language) is easy. We use a hypothetical language based on Knuth's MIX [6] because its features are similar to most well-known machines and its inherent simplicity allows us to write programs in clear and concise form.

Program C below is a MIX-like implementation of Algorithm C. Liberties have been taken with the language for purposes of clarity; the actual MIX code appears in [10]. The program is written in a five-column format: the first column gives the line numbers, the second column lists the instruction labels, the third column contains the assembly language instructions, the fourth column counts the number of times the instructions are executed, and the last column is for comments that explain what the instructions do. The syntax of the commands should be clear to those familiar with assembly language programming. The four memory registers

used in Program C are named rA, rX, rI, and rJ. The reference KEY(I) denotes the contents of the memory location whose address is the value of KEY plus the contents of rI. (This is $KEY[i]$ in the notation of Algorithm C.)

Program C (*Coalesced hashing search and insertion*). This program follows the conventions of Algorithm C, except that the *EMPTY* field is implicit in the *LINK* field: empty slots are marked by a -1 in the *LINK* field of that slot. Null links are denoted by a 0 in the *LINK* field. The variable R and the key K are stored in memory locations R and K. Registers rI and rA are used to store the values of i and K. Register rJ stores either the value of $LINK[i]$ or R. The instruction labels SUCCESS and OVERFLOW are for exiting and are assumed to lie somewhere outside this code.

01	START	LD	X, K	1	*Step C1.* Load rX with K.
02		ENT	A, 0	1	Enter 0 into rA.
03		DIV	=M=	1	rA $\leftarrow \lfloor K/M \rfloor$, rX $\leftarrow K \bmod M$.
04		ENT	I, X	1	Enter rX into rI.
05		INC	I, 1	1	Increment rI by 1.
06		LD	A, K	1	Load rA with K.
07		LD	J, LINK(I)	1	*Step C2.* Load rJ with $LINK[i]$.
08		JN	J, STEP6	1	Jump to STEP6 if $LINK[i] < 0$.
09		CMP	A, KEY(I)	A	*Step C3.* Compare K with $KEY[i]$.
10		JE	SUCCESS	A	Exit (successessfully) if $K = KEY[i]$.
11		JZ	J, STEP5	$A - S1$	Jump to STEP5 if $LINK[i] = 0$.
12	STEP4	ENT	I, J	$C - 1$	*Step C4.* Enter rJ into rI.
13		CMP	A, KEY(I)	$C - 1$	*Step C3.* Compare K with $KEY[i]$.
14		JE	SUCCESS	$C - 1$	Exit (successessfully) if $K = KEY[i]$.
15		LD	J, LINK(I)	$C - 1 - S2$	Load rJ with $LINK[i]$.
16		JNZ	J, STEP4	$C - 1 - S2$	Jump to STEP4 if $LINK[i] \neq 0$.
17	STEP5	LD	J, R	$A - S$	*Step C5.* Load rJ with R.
18		DEC	J, 1	T	Decrement R by 1.
19		LD	X, LINK(J)	T	Load rX with $LINK[R]$.
20		JNN	X, *-2	T	Go back two steps if $LINK[R] \geq 0$.
21		JZ	J, OVERFLOW	$A - S$	Exit (with overflow) if $R = 0$.
22		ST	J, LINK(I)	$A - S$	Store R in $LINK[i]$
23		ENT	I, J	$A - S$	Enter rJ into rI.
24		ST	J, R	$A - S$	Update R in memory.
25	STEP6	ST	0, LINK(I)	$1 - S$	*Step C6.* Store 0 in $LINK[i]$.
26		ST	A, KEY(I)	$1 - S$	Store K in $KEY[i]$. ■

The execution time is measured in MIX *units of time*, which we denote u. The number of time units required by an instruction is equal to the number of memory references (including the reference to the instruction itself). Hence, the LD, ST, and CMP instructions each take two units of time, while ENT, INC, DEC, and the jump instructions require only one time unit. The division operation used to compute the hash address is an exception to this rule; it takes $14u$ to execute.

The running time of a MIX program is the weighted sum

$$\sum_{\substack{\text{each instruction}\\ \text{in the program}}} \begin{pmatrix} \#\text{ times} \\ \text{the instruction} \\ \text{is executed} \end{pmatrix} \begin{pmatrix} \#\text{ time units} \\ \text{required by} \\ \text{the instruction} \end{pmatrix} \quad (1)$$

This is a somewhat simplistic model, since it does not make use of cache or buffered memory for fast access of frequently used data, and since it ignores any intervention by the operating system. But it places all hashing algorithms on an equal footing and gives a good indication of relative merit.

The fourth column of Program C expresses the number of times each instruction is executed in terms of the quantities

C = number of probes per search.
$A = 1$ if the initial probe found an occupied slot, 0 otherwise.
$S = 1$ if successful, 0 if unsuccessful.
T = number of slots probed while looking for an empty space.

We further decompose S into $S1 + S2$, where $S1 = 1$ if the search is successful on the first probe, and $S1 = 0$ otherwise. By formula (1), the total running time of the searching phase is

$$(7C + 4A + 17 - 3S + 2S1)u \quad (2)$$

and the insertion of a new record after an unsuccessful search (when $S = 0$) takes an additional $(8A + 4T + 4)u$. The average running time is the expected value of (2), assuming that all insertions and searches are random. The formula can be obtained by replacing the variables in Eq. (2) with their expected values.

4. Tuning β to Obtain Optimum Performance

The purpose of the analysis in [10, 11, 13] is to show how the average-case performance of the coalesced hashing method varies as a function of the address factor $\beta = M/M'$ and the load factor $\alpha = N/M'$. In this section, for each *fixed* value of α, we make use of those results in order to "tune" our choice of β and speed up the search times. Our two measures of performance are the expected number of probes per search and the average running time of assembly language versions. In the latter case, we study a MIX implementation in detail, and then show how to apply what we learn to other assembly languages.

Unfortunately, there is no single choice of β that yields best results: the optimum choice β_{OPT} is a function of the load factor α and it is even different for unsuccessful and successful searches. The section concludes with practical tips on how to initialize β. In particular, we shall see that the choice $\beta \approx 0.86$ works well in most situations.

4.1 Number of Probes Per Search

For each fixed value of α, we want to find the values β_{OPT} that minimize the expected number of search probes in unsuccessful and successful searches. Formulas (A1) and (A2) in the Appendix express the average number of probes per search as a function of three variables: the load factor $\alpha = N/M'$, the address factor $\beta = M/M'$, and a new variable $\lambda = L/M$, where L is the expected number of inserted records needed to make the cellar become full. The variables β and λ are related by the formula

$$e^{-\lambda} + \lambda = \frac{1}{\beta} \qquad (3)$$

Formulas (A1) and (A2) each have two cases, "$\alpha \leq \lambda\beta$" and "$\alpha \geq \lambda\beta$," which have the following intuitive meanings: The condition $\alpha < \lambda\beta$ means that with high probability not enough records have been inserted to fill up the cellar, while the condition $\alpha > \lambda\beta$ means that enough records have been inserted to make the cellar almost surely full.

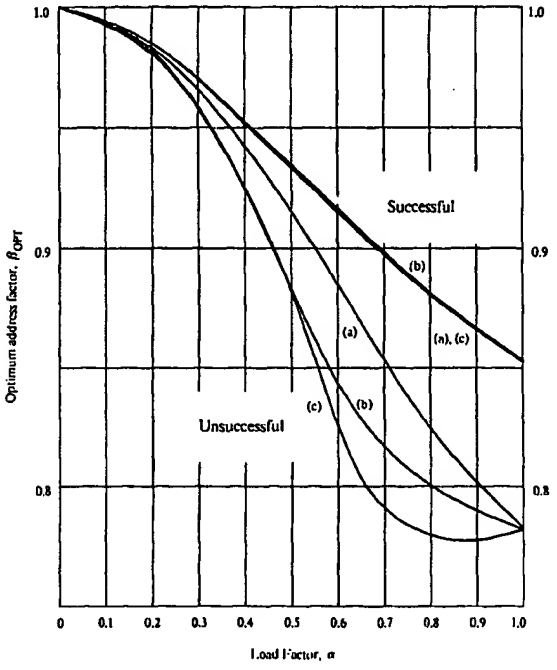
The optimum address factor β_{OPT} is always located somewhere in the "$\alpha \geq \lambda\beta$" region, as shown in the Appendix. The rest of the optimization procedure is a straightforward application of differential calculus. First, we substitute Eq. (3) into the "$\alpha \geq \lambda\beta$" cases of the formulas for the expected number of probes per search in order to express them in terms of only the two variables α and λ. For each nonzero fixed value of α, the formulas are convex w.r.t. λ and have unique minima. We minimize them by setting their derivatives equal to 0. Numerical analysis techniques are used to solve the resulting equations and to get the optimum values of λ for several different values of α. Then we reapply Eq. (3) to express the optimum points in terms of β. The results are graphed in Fig. 2(a), using spline interpolation to fill in the gaps.

4.2 MIX Running Times

Optimizing the MIX execution times could be tricky, in general, because the formulas might have local as well as global minima. Then when we set the derivatives equal to 0 in order to find β_{OPT}, there might be several roots to the resulting equations. The crucial fact that lets us apply the same optimization techniques we used above for the number of probes is that the formulas for the MIX running times are *well-behaved*, as shown in the Appendix. By that we mean that each formula is minimized at a *unique* β_{OPT}, which occurs either at the endpoint $\alpha = \lambda\beta$ or at the unique point in the "$\alpha \geq \lambda\beta$" region where the derivative w.r.t. β is 0.

The optimization procedure is the same as before. The expected values of formulas (A4) and (A5), which give the MIX running times for unsuccessful and successful searches, are functions of the three variables α, β, and λ. We substitute Eq. (3) into the expected running times in order to express β in terms of λ. For several different load factors α and for each type of search, we find the value of λ that minimizes the formula, and then we retranslate this value via Eq. (3) to get β_{OPT}. Figure 2(b) graphs these optimum values β_{OPT} as a function of α; spline interpolation was used to fill in the gaps. As in the previous section, the formulas for the average unsuccessful and successful search times yield different optimum address factors. For the successful search case, notice how closely β_{OPT} agrees with the corresponding values that minimize the expected number of probes.

Fig. 2. The values β_{OPT} that optimize search performance for the following three measures: (a) the expected number of probes per search, (b) the expected running time of Program C, and (c) the expected assembly language running time for large keys.



4.3 Applying the Results to Other Implementations

Our MIX analysis suggests two important principles to be used in finding β_{OPT} for a particular implementation of coalesced hashing. First, the formulas for the expected number of times each instruction in the program is executed (which are expressed for Program C in terms of C, A, S, $S1$, $S2$, and T) may have the two cases, "$\alpha \leq \lambda\beta$" and "$\alpha \geq \lambda\beta$," but probably not more.

Second, the same optimization process as above can be used to find β_{OPT}, because the formulas for the running times should be well-behaved for the following reason: The main difference between Program C and another implementation is likely to be the relative time it takes to process each key. (The keys are assumed to be very small in the MIX version.) Thus, the unsuccessful search time for another implementation might be approximately

$$[(2\kappa + 5)C + (2\kappa + 2)A + (-2\kappa + 19)]u' \qquad (4)$$

where u' is the standard unit of time on the other computer and κ is how many times longer it takes to process a key (multiplied by u/u'). Successful search times would be about

$$[(2\kappa + 5)C + 18 + 2S1]u' \qquad (5)$$

Formulas (4) and (5) were calculated by increasing the execution times of the key-processing steps 9 and 13 in Program C by a factor of κ. (See formulas (A4) and (A5) for the $\kappa = 1$ case.) We ignore the extra time it takes to load the larger key and to compute the hash function, since that does not affect the optimization.

The role of C in formula (4) is less prevalent than in (A4) as κ gets large: the ratio of the coefficients of C and A decreases from 7/4 in (A4) and approaches the limit $2/2 = 1$ in formula (4). Even in this extreme case, however, computer calculations show that the formula for the average running time is well-behaved. The values of β_{OPT} that minimize formula (4) when κ is large are graphed in Fig. 2(c).

For successful searches, however, the value of C more strongly *dominates* the running times for larger values of κ, so the limiting values of β_{OPT} in Fig. 2(c) *coincide* with the ones that minimize the expected number of probes per search in Fig. 2(a). Figure 2(b) shows that the approximation is close even for the case $\kappa = 1$, which is Program C.

4.4 How to Choose β

It is important to remember that the address region size $M = \lceil \beta M' \rceil$ must be initialized when the hash table is empty and cannot change thereafter. Unfortunately, the last two sections show that each different load factor α requires a different optimum address factor β_{OPT}; in fact, the values of β_{OPT} differ for unsuccessful and successful searches. This means that optimizing the average unsuccessful (or successful) search time for a certain load factor α will lead to suboptimum performance when the load factor is not equal to α.

One strategy is to pick $\beta \approx 0.782$, which minimizes the expected number of probes per unsuccessful search as well as the average MIX unsuccessful search time when the table is full (i.e., load factor $\alpha = 1$), as indicated in Fig. 2. This choice of β yields the best absolute bound on search performance, because when the table is full, search times are greatest and unsuccessful searches average slightly longer than successful ones. Regardless of the load factor, the expected number of probes per search would be at most 1.79, and the average MIX searching time would be bounded by $33.52u$.

Another strategy is to pick some compromise address factor that leads to good overall performance for a *large range* of load factors. A reasonable choice is $\beta = 0.86$; then the unsuccessful searches are optimized (over all other values of β) when the load factor is $\approx$0.68 (number of probes) and $\approx$0.56 (MIX), and the successful search performance is optimized at load factors $\approx$0.94 (number of probes) and $\approx$0.95 (MIX).

Figures 3 through 6 graph the expected search performance of coalesced hashing as a function of α for both types of searches (unsuccessful and successful) and for both measures of performance (number of probes and MIX running time). The C_1 curve corresponds to standard coalesced hashing (i.e., $\beta = 1$); the $C_{0.86}$ line is our compromise choice $\beta = 0.86$; and the dashed line C_{OPT} represents the best possible search performance that could be achieved by tuning (in which β is optimized for each load factor).

Notice that the value $\beta = 0.86$ yields near-optimum search times once the table gets half-full, so this compromise offers a viable strategy. Of course, if some prior knowledge about the types and frequencies of the searches were available, we could tailor our choice of β to meet those specific needs.

5. Comparisons

In this section, we compare the searching times of the coalesced hashing method with those from a representative collection of hashing schemes: standard coalesced hashing (C_1), separate chaining (S), separate chaining with ordered chains (SO), linear probing (L), and double hashing (D). Implementations of the methods are given in [10].

These methods were chosen because they are the most well-known and since they each have implementations similar to that of Algorithm C. Our comparisons are based both on the expected number of probes per search as well as on the average MIX running time. Coalesced hashing performs better than the other methods. The differences are not so dramatic with the MIX search times as with the number of probes per search, due to the large overhead in computing the hash address. However, if the keys were larger and comparisons took longer, the relative MIX savings would closely approximate the savings in number of probes.

Fig. 3. The average number of probes per unsuccessful search, as M and $M' \to \infty$, for coalesced hashing (C_1, $C_{0.86}$, C_{OPT} for $\beta = 1$, 0.86, β_{OPT}), separate chaining (S), separate chaining with ordered chains (SO), linear probing (L), and double hashing (D).

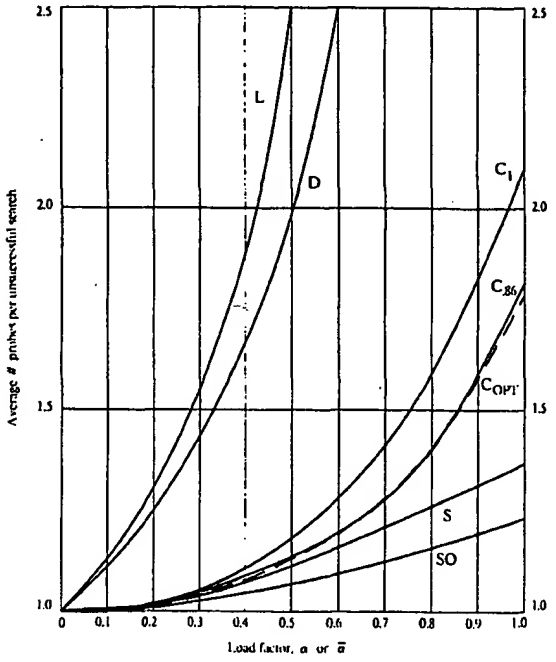
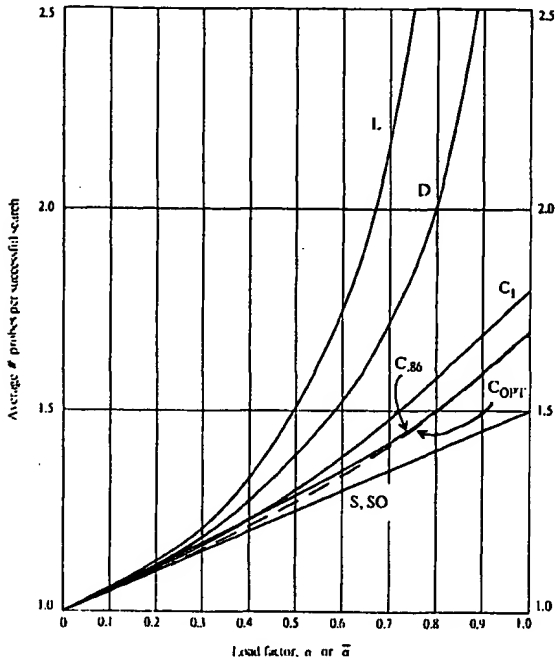


Fig. 4. The average number of probes per successful search, as M and $M' \to \infty$, for coalesced hashing (C_1, $C_{0.86}$, C_{OPT} for $\beta = 1$, 0.86, β_{OPT}), separate chaining (S), separate chaining with ordered chains (SO), linear probing (L), and double hashing (D).



5.1 Standard Coalesced Hashing (C_1)

Standard coalesced hashing is the special case of coalesced hashing for which $\beta = 1$ and there is no cellar. This is obviously the most realistic comparison that can be made, because except for the initialization of the address region size, standard coalesced hashing and "tuned" coalesced hashing are identical. Figures 3 and 4 show that the savings in number of probes per search can be as much as 14 percent (unsuccessful) and 6 percent (successful). In Figs. 5 and 6, the corresponding savings in MIX searching time is 6 percent (unsuccessful) and 2 percent (successful).

Fig. 5. The average MIX execution time per unsuccessful search, as $M' \to \infty$, for coalesced hashing (C_1, $C_{0.86}$, C_{OPT} for $\beta = 1$, 0.86, β_{OPT}), separate chaining (S), separate chaining with ordered chains (SO), linear probing (L), and double hashing (D).

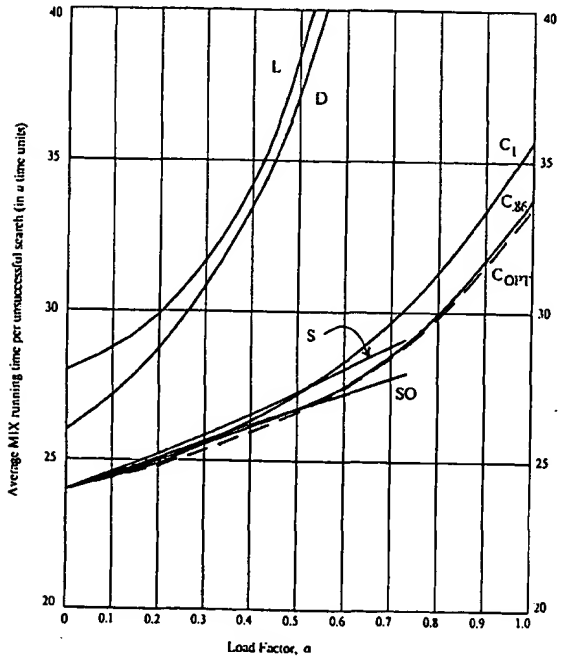
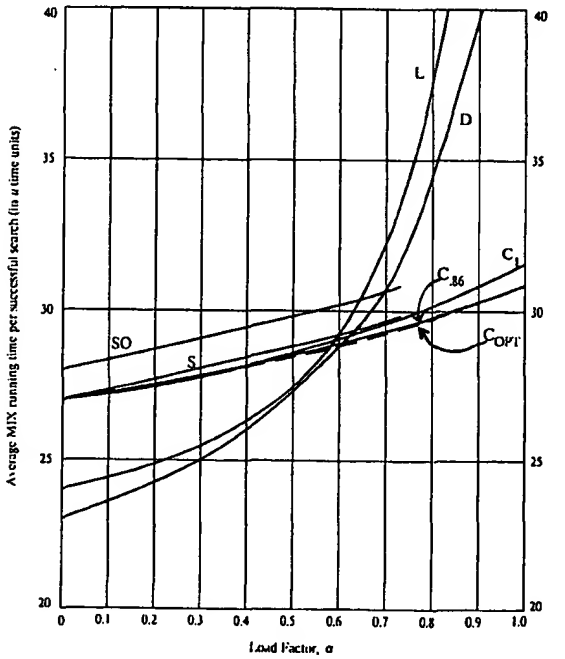


Fig. 6. The average MIX execution time per successful search, as $M' \to \infty$, for coalesced hashing (C_1, $C_{0.86}$, C_{OPT} for $\beta = 1$, 0.86, β_{OPT}), separate chaining (S), separate chaining with ordered chains (SO), linear probing (L), and double hashing (D).



5.2 Separate (or Direct) Chaining (S)

The separate chaining method is given an unfair advantage in Figs. 3 and 4: the number of probes per search is graphed as a function of $\bar{\alpha} = N/M$ rather than $\alpha = N/M'$ and does not take into account the number of auxiliary slots used to store colliders. In order to make the comparison fair, we must adjust the load factor accordingly.

Separate chaining implementations are designed often to accommodate about $N = M$ records; an average of $M(1 - 1/M)^M \approx M/e$ auxiliary slots are needed to store the colliders. The total table size is thus $M' \approx M + M/e$. Solving backwards for M, we get $M \approx 0.731M'$. In other words, we may consider separate chaining to be the special case of coalesced hashing for which $\beta \approx 0.731$, except that no more records can be inserted once the cellar overflows. Hence, the adjusted load factor is $\alpha \approx 0.731\bar{\alpha}$, and overflow occurs when there are around $N = M \approx 0.731M'$ inserted records. (This is a reasonable space/time compromise: if we make M smaller, then more records can usually be stored before overflow occurs, but the average search times blow up; if we increase M to get better search times, then overflow occurs much sooner, and many slots are wasted.)

If we adjust the load factors in Figs. 3 and 4 in this way, Algorithm C generates better search statistics: the expected number of probes per search for separate chaining is ≈ 1.37 (unsuccessful) and ≈ 1.5 (successful) when the load factor $\bar{\alpha}$ is 1, while that for coalesced hashing is ≈ 1.32 (unsuccessful) and ≈ 1.44 (successful) when the load factor $\alpha = \beta\bar{\alpha}$ is equal to 0.731.

The graphs in Figs. 5 and 6 already reflect this load factor adjustment. In fact, the MIX implementation of separate chaining (Program S in [10]) is identical to Program C, except that β is initialized to 0.731 and overflow is signaled automatically when the cellar runs out of empty slots. Program C is slightly quicker in MIX execution time than Program S, but more importantly, the coalesced hashing implementation is more space efficient: Program S usually overflows when $\alpha \approx 0.731$, while Program C can always obtain full storage utilization $\alpha = 1$. This confirms our intuition that coalesced hashing can accomodate more records than the separate chaining method and still outperform separate chaining before that method overflows.

5.3 Separate Chaining with Ordered Chains (SO)

This method is a variation of separate chaining in which the chains are kept ordered by key value. The expected number of probes per successful search does not change, but unsuccessful searches are slightly quicker, because only about half the chain needs to be searched, on the average.

Our remarks about adjusting the load factor in Figs. 3 and 4 also apply to method SO. But even after that is done, the average number of probes per unsuccessful search as well as the expected MIX unsuccessful search time is slightly better for this method than for coalesced hashing. However, as Fig. 6 illustrates, the average successful search time of Program SO is worse than Program C's, and in real-life situations, the difference is likely to be more apparent, because records that are inserted first tend to be looked up more often and should be kept near the beginning of the chain, not rearranged.

Method SO has the same storage limitations as the separate chaining scheme (i.e., the table usually overflows when $N \approx M \approx 0.731M'$), whereas coalesced hashing can obtain full storage utilization.

5.4 Linear Probing (L) and Double Hashing (D)

When searching for a record with key K, the linear probing method first checks location $hash(K)$, and if another record is already there, it steps cyclically through the table, starting at location $hash(K)$, until the record is found (successful search) or an empty slot is reached (unsuccessful search). Insertions are done by placing the record into the empty slot that terminated the unsuccessful search. Double hashing generalizes this by letting the cyclic step size be a function of K.

We have to adjust the load factor in the *opposite* direction when we compare Algorithm C with methods L and D, because the latter do not require *LINK* fields. For example, if we suppose that the *LINK* field comprises $\frac{1}{4}$ of the total record size in a coalesced hashing implementation, then the search statistics in Figs. 3 and 4 for Algorithm C with load factor α should be compared against those for linear probing and double hashing with load factor $(\frac{3}{4})\alpha$. In this case, the average number of probes per search is still better for coalesced hashing.

However, the *LINK* field is often much smaller than the rest of the record, and sometimes it can be included in the table at virtually no extra cost. The MIX implementation Program C in [10] assumes that the MIX field can be squeezed into the record without need of extra storage space. Figures 5 and 6, therefore, require no load factor adjustment.

To balance matters, the MIX implementations of linear probing and double hashing, which are given in [10] and [7], contain two code optimizations. First, since *LINK* fields are not used in methods L and D, we no longer need 0 to denote a null *LINK*, and we can renumber the table slots from 0 to $M' - 1$; the hash function now returns a value between 0 and $M' - 1$. This makes the hash address computation faster by $1u$, because the instruction INC I, 1 can be eliminated. Second, the empty slots are denoted by the value 0 in order to make the comparisons in the inner loop as fast as possible. This means that records are not allowed to have a key value of 0. The final results are graphed in Figs. 5 and 6. Coalesced hashing clearly dominates when the load factor is greater than 0.6.

6. Deletions

It is often useful in hashing applications to be able to delete records when they no longer logically belong to the set of objects being represented in the hash table. For

example, in an airlines reservations system, passenger records are often expunged soon after the flight has taken place.

One possible deletion strategy often used for linear probing and double hashing is to include a special one-bit *DELETED* field in each record that says whether or not the record has been deleted. The search algorithm must be modified to treat each "deleted" table slot as if it were occupied by a null record, even though the entire record is still there. This is especially desirable when there are pointers to the records from *outside* the table.

If there are no such external pointers to worry about, the "deleted" table slots can be reused for later insertions: Whenever an empty slot is needed in step C5 of Algorithm C, the record is inserted into the first "deleted" slot encountered during the unsuccessful search; if there is no such slot, an empty slot is allocated in the usual way. However, a certain percentage of the "deleted" slots probably will remain unused, thus preventing full storage utilization. Also, insertions and deletions over a prolonged period would cause the expected search times to approximate those for a full table, regardless of the number of undeleted records, because the "deleted" records make the searches longer.

If we are willing to spend a little extra time per deletion, we can do without the *DELETED* field by relocating some of the records that follow in the chain. The basic idea is this: First, we find the record we want to delete, mark its table slot empty, and set the *LINK* field of its predecessor (if any) to the null value 0. Then we use Algorithm C to reinsert each record in the remainder of the chain, but whenever an empty slot is needed in step C5, we use the position that the record already occupies.

This method can be illustrated by deleting AL from location 10 in Fig. 7(a); the end result is pictured in Fig. 7(b). The first step is to create a hole in position 10 where AL was, and to set AUDREY's *LINK* field to 0. Then we process the remainder of the chain. The next record TOOTIE rehashes to the hole in location 10, so TOOTIE moves up to plug the hole, leaving a new hole in position 9. Next, DONNA collides with AUDREY during rehashing, so DONNA remains in slot 8 and is linked to AUDREY. Then MARK also collides with AUDREY; we leave MARK in position 7 and link it to DONNA, which was formerly at the end of AUDREY's hash chain. The record JEFF rehashes to the hole in slot 9, so we move it up to plug the hole, and a new hole appears in position 6. Finally, DAVE rehashes to position 9 and joins JEFF's chain.

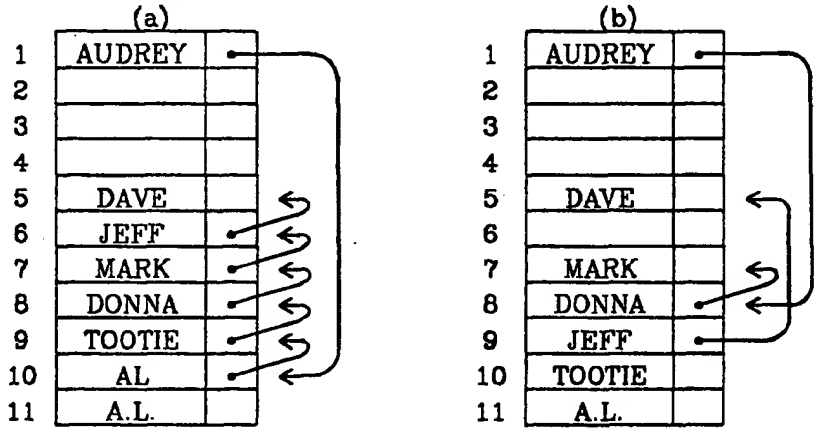
Location 6 is the current hole position when the deletion algorithm terminates, so we set $EMPTY[6] \leftarrow$ **true** and return it to the pool of empty slots. However, the value of R in Algorithm C is already 5, so step C5 will never try to reuse location 6 when an empty slot is needed.

We can solve this problem by using an *available-space list* in step C5 rather than the variable R; the list must be doubly linked so that a slot can be removed quickly from the list in step C6. The available-space list does not require any extra space per table slot, since we can use the *KEY* and *LINK* fields of the empty slots for the two pointer fields. (The *KEY* field is much larger than the *LINK* field in typical implementations.) For clarity, we rename the two pointer fields *NEXT* and *PREV*. Slot 0 in the table acts as the dummy start of the available-space list, so $NEXT[0]$ points to the first actual slot in the list and $PREV[0]$ points to the last. Before any records are inserted into the table, the following extra initializations must be made: $NEXT[0] \leftarrow M'$ $PREV[M'] \leftarrow 0$; and $NEXT[i] \leftarrow i - 1$ and $PREV[i - 1] \leftarrow i$, for $1 \le i \le M'$. We replace steps C5 and C6 by

C5. [Find empty slot.] (The search for K in the chain was unsuccessful, so we will try to find an empty table slot to store K.) If the table is already full (i.e., $NEXT[0] = 0$), the algorithm terminates *with overflow*. Otherwise, set $LINK[i] \leftarrow NEXT[0]$ and $i \leftarrow NEXT[0]$.

C6. [Insert new record.] Remove the ith slot from the

Fig. 7. (a) Inserting the eight records; (b) Inserting all the records *except* AL.



Keys:	A.L.	AUDREY	AL	TOOTIE	DONNA	MARK	JEFF	DAVE
Hash Addresses:	11	1	1	10	1	1	9	9

available-space list by setting $PREV[NEXT[i]] \leftarrow PREV[i]$ and $NEXT[PREV[i]] \leftarrow NEXT[i]$. Then set $EMPTY[i] \leftarrow$ **false**, $KEY[i] \leftarrow K$, $LINK[i] \leftarrow 0$, and initialize the other fields in the record.

The following deletion algorithm is analyzed in detail in [10] and [14].

Algorithm CD (*Deletion with coalesced hashing*). This algorithm deletes the record with key K from a coalesced hash table constructed by Algorithm C, with steps C5 and C6 modified as above.

This algorithm preserves the important invariant that K is stored at its hash address if and only if it is at the start of its chain. This makes searching for K's predecessor in the chain easy: if it exists, then it must come *at* or *after* position $hash(K)$ in the chain.

CD1. [Search for K.] Set $i \leftarrow hash(K)$. If $EMPTY[i]$, then K is not present in the table and the algorithm terminates. Otherwise, if $K = KEY[i]$, then K is at the start of the chain, so go to step CD3.

CD2. [Split chain in two.] (K is not at the start of its chain.) Repeatedly set $PRED \leftarrow i$ and $i \leftarrow LINK[i]$ until either $i = 0$ or $K = KEY[i]$. If $i = 0$, then K is not present in the table, and the algorithm terminates. Else, set $LINK[PRED] \leftarrow 0$.

CD3. [Process remainder of chain.] (Variable i will walk through the successors of K in the chain.) Set $HOLE \leftarrow i$, $i \leftarrow LINK[i]$, $LINK[HOLE] \leftarrow 0$. Do step CD4 zero or more times until $i = 0$. Then go to step CD5.

CD4. [Rehash record in ith slot.] Set $j \leftarrow hash(KEY[i])$. If $j = HOLE$, we move up the record to plug the hole by setting $KEY[HOLE] \leftarrow KEY[i]$ and $HOLE \leftarrow i$. Otherwise, we link the record to the end of its hash chain by doing the following: set $j \leftarrow LINK[j]$ zero or more times until $LINK[j] = 0$; then set $LINK[j] \leftarrow i$. Set $k \leftarrow LINK[i]$, $LINK[i] \leftarrow 0$, and $i \leftarrow k$. Repeat step CD4 unless $i = 0$.

CD5. [Mark slot $HOLE$ empty.] Set $EMPTY[HOLE] \leftarrow$ **true**. Place $HOLE$ at the start of the available-space list by setting $NEXT[HOLE] \leftarrow NEXT[0]$, $PREV[HOLE] \leftarrow 0$, $PREV[NEXT[0]] \leftarrow HOLE$, $NEXT[0] \leftarrow HOLE$. ■

Algorithm CD has the important property that it *preserves randomness* for the special case of standard coalesced hashing (when $M = M'$), in that deleting a record is in some sense like never having inserted it. The "sense" is strong enough so that the formulas for the average search times are still valid after deletions are performed. Exactly what preserving randomness means is explained in detail in [14].

We can speed up the rehashing phase in the latter half of step CD4 by linking the record into the chain *immediately after* its hash address rather than at the end of the chain. When this modified deletion algorithm is called on a random standard coalesced hash table, the resulting table is *better-than-random*: the average search times after N random insertions and one deletion are sometimes better (and never worse) than they would be with $N - 1$ random insertions alone. Whether or not this remains true after more than one deletion is an open problem.

If this deletion algorithm is used when there is a cellar (i.e., $\beta < 1$), we can modify it so that whenever a hole appears in the cellar during the execution of Algorithm CD, the next noncellar record in the chain moves up to plug the hole. Unfortunately, even with this modification, the algorithm does not break up chains well enough to preserve randomness. It seems possible that search performance may remain very good anyway. Analytic and empirical study is needed to determine just "how far from random" the search times get after deletions are performed.

Two remarks should be made about implementing this modified deletion algorithm. In step CD6, the empty slot should be returned to the start of the available-space list when the slot is in the cellar; otherwise, it should be placed at the *end*. This has the effect of giving cellar slots higher priority on the available-space list. Second, if a cellar slot is freed by a deletion and then reallocated during a later insertion, it is possible for chain to go in and out of the cellar more than once. Programmers should no longer assume that a chain's cellar slots immediately follow the start of the chain.

7. Implementations and Variations

Most important searching algorithms have several different implementations in order to handle a variety of applications; coalesced hashing is no exception. We have already discussed some modifications in the last section in connection with deletion algorithms. In particular, we needed to use a doubly linked available-space list so that the empty slots could be added and removed quickly. Thus, the cellar need not be contiguous. Another strategy to handle a noncontiguous cellar is to link all the table slots together initially and to replace "Decrease R" in step C5 of Algorithm C with "Set $R \leftarrow LINK[R]$." With either modification, Algorithm C can simulate the separate chaining method until the cellar empties; subsequent colliders can be stored in the address region as usual. Hence, coalesced hashing can have the benefit of dynamic allocation as well as total storage utilization.

Another common data structure is to store pointers to the fields, rather than the fields themselves, in the table slots. For example, if the records are large, we might want to store only the key and link values in each slot, along with a pointer to where the rest of the record is located. We expand upon this idea later in this section.

If we are willing to do extra work during insertion and if the records are not pointed to from outside the table, we can modify the insertion algorithm to prevent the chains from coalescing: When a record R_1 collides during insertion with another record R_2 that is *not* at the

start of the chain, we store R_1 at its hash address and relocate R_2 to some other spot. (The *LINK* field of R_2's predecessor must be updated.) The size of the records should not be very large or else the cost of rearrangement might get prohibitive. There is an alternate strategy that prevents coalescing and does not relocate records, but it requires an extra link field per slot and the searches are slightly longer. One link field is used to chain together all the records with the same hash address. The other link field contains for slot i a pointer to the start of the chain of records with hash address i. Much of the space for the link fields is wasted, and chains may start one link away from their hash address. Resources could be put to better use by using coalesced hashing.

This section is devoted to the more nonobvious implementations of coalesced hashing. First, we describe alternate insertion strategies and then conclude with three applications to external searching on secondary storage devices. A scheme that allows the coalesced hash table to share memory with other data structures can be found in [12]. A generalization of coalesced hashing that uses nonuniform hash functions is described in [13].

7.1 Early-Insertion and Varied-Insertion Coalesced Hashing

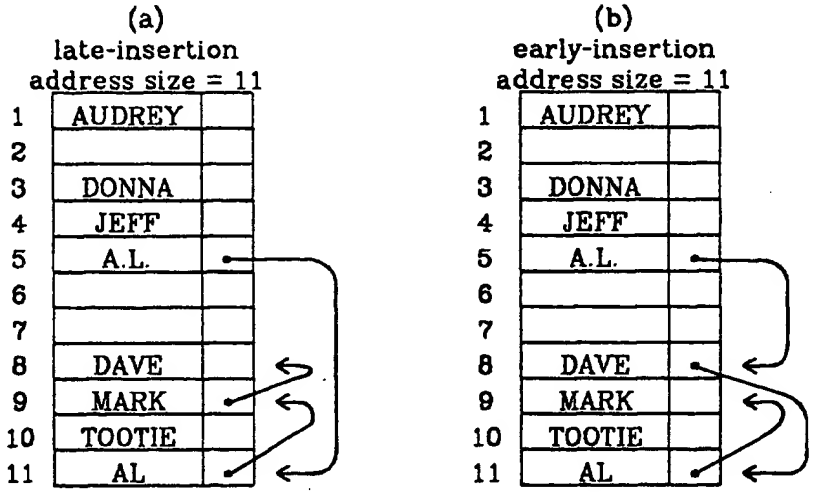
If we know *a priori* that a record is not already present in the table, then it is not necessary in Algorithm C to search to the end of the chain before the record is inserted: If the hash address location is empty, the record can be inserted there; otherwise, we can link the record into the chain *immediately after* its hash address by rerouting pointers. We call this the *early-insertion* method because the collider is linked "early" in the chain, rather than at the end. We will refer to the unmodified algorithm (Algorithm C in Sec. 2) as the *late-insertion* method.

Early-insertion can be used even if we do not have *a priori* knowledge about the record's presence, in which case the entire chain must be searched in order to verify that the record is not already stored in the table. We can implement this form of early-insertion by making the following two modifications to Algorithm C. First, we add the assignment "Set $j \leftarrow i$" at the end of step C2, so that j stores the hash address $hash(K)$. The second modification replaces the last sentence of step C5 by "Otherwise, link the Rth cell into the chain immediately after the hash address j by setting $LINK[R] \leftarrow LINK[j]$, $LINK[j] \leftarrow R$; then set $i \leftarrow R$."

Each chain of records formed using early-insertion contains the same records as the corresponding chain formed by late-insertion. Since the length of a random unsuccessful search depends only on the number of records in the chain between the hash address and the end of the chain, and since all the records are in the address region when there is no cellar, it must be true that the average number of probes per unsuccessful search is the same for the two methods if there is no cellar. However, the order of the records within each chain may be different for early-insertion than for late-insertion. When there is no cellar, the early-insertion algorithm causes the records to align themselves in the chains closer to their hash addresses, on the average, than would be the case with late-insertion, so the expected successful search times are better.

A typical case is illustrated in Fig. 8. The record DAVE collides with A.L. at slot 5. In Fig. 8(a), which uses late-insertion, DAVE is linked to the end of the chain containing A.L., whereas if we use early-insertion as in Fig. 8(b),

Fig. 8. Standard Coalesced Hashing, $M = M' = 11$, $N = 8$. (a) Late-insertion; (b) Early-insertion.



Keys:	A.L.	AUDREY	AL	TOOTIE	DONNA	MARK	JEFF	DAVE
Hash Addresses:	5	1	5	10	3	11	4	5

ave. # probes per succ. search: (a) 13/8 ≈ 1.63, (b) 12/8 = 1.5.

DAVE is linked into the chain at the point between A.L. and AL. The average successful search time in Fig. 8(b) is slightly better than in Fig. 8(a), because linking DAVE into the chain immediately after A.L. (rather than at the end of the chain) reduces the search time for DAVE from four probes to two and increases the search time for AL from two probes to three. The result is a net decrease of one probe.

One can show easily that this effect manifests itself only on chains of length greater than 3, so there is little improvement when the load factor α is small, since the chains are usually short. Recent theoretical results show that the average number of probes per successful search is 5 percent better with early-insertion than with late-insertion when there is no cellar and the table is full (i.e., $\alpha = 1$), but is only 0.5 percent better when $\alpha = 0.5$ [1, 5]. A possible disadvantage of early-insertion is that earlier colliders tend to be shoved to the rear by later ones, which may not be desirable in some practical situations when the records inserted first tend to be accessed more often than those inserted later. Nevertheless, early-insertion is an improvement over late-insertion when there is no cellar.

When there is a cellar, preliminary studies indicate that search performance is probably worse with early-insertion than with Algorithm C, because a chain's records that are in the cellar now come at the *end* of the chain, whereas with late-insertion they come immediately after the start. In the example in Fig. 9(b), the insertion of JEFF causes both cellar records AL and TOOTIE to move one link further from their hash addresses. That does not happen with late-insertion in Fig. 9(b).

We shall now introduce a new variant, called *varied-insertion*, that can be shown to be better than both the late-insertion and early-insertion methods when there is a cellar. When there is no cellar, varied-insertion is identical to early-insertion. In the varied-insertion method, the early-insertion strategy is used *except* when the cellar is full and the hash address of the inserted record is the start of a chain that has records in the cellar. In that case, the record is linked into the chain *immediately after the last cellar slot in the chain.*

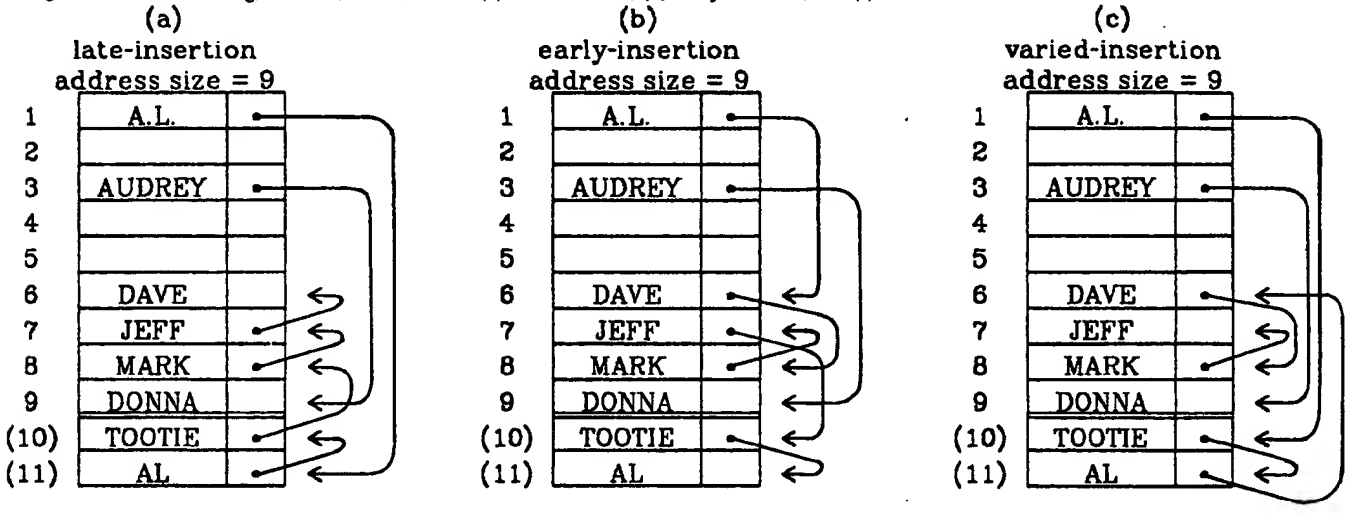
Figure 9(c) shows a typical hash table constructed using varied-insertion. The cellar is already full when the record DAVE is inserted. The hash address of DAVE is 1, which is at the start of a chain that has records in the cellar. Therefore, early-insertion is not used, and DAVE is instead linked into the chain immediately after AL, which is the last record in the chain that is in the cellar. The average number of probes per search is better for varied-insertion than for both late-insertion and early-insertion.

The varied-insertion method incorporates the advantages of early-insertion, but without any of the drawbacks described three paragraphs earlier. The records of a chain that are in the cellar always come immediately after the start of the chain. The average number of probes per search for varied-insertion is always less than or equal to that for late-insertion and early-insertion. For unsuccessful searches, the expected number of probes for varied-insertion and late-insertion are identical.

Research is currently underway to determine the average search times for the varied-insertion method, as well as to find the values of the optimum address factor β_{OPT}. We expect that the initialization $\beta \leftarrow 0.86$ will be preferred in most situations, as it is for late-insertion. The resulting search times for varied-insertion should be a slight improvement over late-insertion.

The idea of linking the inserted record into the chain immediately after its hash address has been incorporated into the first modification of Algorithm CD in the last

Fig. 9. Coalesced Hashing, $M' = 11$, $M = 9$, $N = 8$. (a) Late-insertion; (b) Early-insertion; and (c) Varied-insertion.



Keys:	A.L.	AUDREY	AL	TOOTIE	DONNA	MARK	JEFF	DAVE
Hash Addresses:	1	3	1	1	3	1	8	1

ave. # probes per unsucc. search: (a) 18/9 = 2.0, (b) 24/9 ≈ 2.67, (c) 18/9 = 2.0.
ave. # probes per succ. search: (a) 21/8 ≈ 2.63, (b) 22/8 = 2.75, (c) 20/8 = 2.5.

section. It is natural to ask whether the modified deletion algorithm would preserve randomness for the modified insertion algorithms presented in this section. The answer is no, but it is possible that the deletion algorithm could make the table better-than-random, as discussed at the end of the last section. Finding good deletion algorithms for early-insertion and varied-insertion as well as for late-insertion is a challenging problem.

7.2 Coalesced Hashing with Buckets

Hashing is used extensively in database applications and file systems, where the hash table is too large to fit entirely in main memory and must be stored on *external* devices, like disks and drums. The hash table is sectioned off into *blocks* (or *pages*), each block containing b records; transfers to and from main memory take place a block at a time. Searching time is dominated by the block transfer rate; now the object is to minimize the expected number of block accesses per search.

Operating systems with a *virtual memory* environment are designed to break up data structures into blocks *automatically*, even though it appears to the programmer that his data structures all reside in main memory. Linear probing (see Sec. 5) is often the best hashing scheme to use in this environment, because successive probes occur in contiguous locations and are apt to be in the same block. Thus, one or two block accesses are usually sufficient for lookup.

We can do better if we know beforehand where the block divisions occur. We treat each block as a large table slot or *bucket* that can store b records. Let M' be the total number of buckets. The following modification of Algorithm C appears in [7].

To process a record with key K, we search for it in the chain of buckets, starting at bucket $hash(K)$. After an unsuccessful search, we insert the record into the last bucket in the chain if there is room, or else we store it in some nonfull bucket and link that bucket to the end of the chain. We can speed up this last part by maintaining a doubly linked circular list of nonfull buckets, with a "roving pointer" marking one of the buckets. Each time we need another nonfull bucket to store a collider, we insert the record into the bucket indicated by the roving pointer, and then we reset the roving pointer to the next bucket on the list. This helps distribute the records evenly, because different chains will use different buckets (at least until we make one loop through the available-bucket list). When the external device is a disk, block accesses are faster when they occur on the same cylinder, so we should keep a separate available-bucket list for each cylinder.

Record size varies from application to application, but for purposes of illustration, we use the following parameters: the block *size* B is 4000 bytes; the total record size R is 400 bytes, of which the key comprises 7 bytes. The bucket size b is approximately $B/R = 10$. When the size of the bucket is that small, searching in each bucket can be done sequentially; there is no need for the record size to be fixed, as long as each record is preceded by its length (in bytes).

Deletions can be done in one of several ways, analogous to the different methods discussed in the last section. In some cases, it is best merely to mark the record as "deleted," because there may be pointers to the record from somewhere outside the hash table, and reusing the space could cause problems. Besides, many large scale database systems undergo periodic reorganization during low-peak hours, in which the entire table (minus the deleted records) is reconstructed from scratch [15]. This method has not been analyzed analytically, but it seems to have great potential.

7.3 Hash Tables Within a Hash Table

When the record size R is small compared to the block size B, the resulting bucket size $b \approx B/R$ is relatively large. Sequential search through the blocks is now too slow. (The block transfer rate no longer dominates search times.) Other methods should be used to organize the records *within* blocks.

This is especially true with *multiattribute* indexing, in which we can look up records via one of several different keys. For example, a large university database may allow a student's record to be accessed by specifying either his name, social security number, student I.D., or bank account number. In this case, *four* hash tables are used. Instead of storing all the records in four different tables, we let the four tables share a single copy of the records. Each hash table entry consists of only the key value, the link field, and a pointer to the rest of the student record (which is stored in some other block). Lookup now requires one extra block access. Continuing our numerical example, the table record size reduces from $R = 400$ bytes to about $R = 12$ bytes, since the key occupies 7 bytes, and the two pointer fields presumably can be squeezed into the remaining 5 bytes. The bucket size b is now about $B/R \approx 333$.

In such cases where b is rather large and searching within a bucket can get expensive, it pays to organize each bucket as a hash table. The hash function must be modified to return a binary number at least $\lceil \log M' \rceil + \lceil \log b \rceil$ bits in length; the high-order bits of the hash address specify one of the M' buckets (or blocks), and the low-order bits specify one of the b record positions within that bucket. Coalesced hashing is a natural method to use because the bucket size (in this example, $b \approx 333$) imposes a definite constraint on the number of records that may be stored in a block, so it is reasonable to try to optimize the amount of space devoted to the address region versus the amount of space devoted to the cellar.

7.4 Dynamic Hashing

So far we have not addressed the problem of what to do when overflow occurs—when we want to insert more records into a hash table that is already full. The common technique is to place the extra records into an auxiliary storage pool and link them to the main table. Search performance remains tolerable as long as the number of insertions after overflow does not get too large. (Guibas [4] analyzes this for the special case of standard coalesced

hashing.) Later during the off-hours when the system is not heavily used, a larger table is allocated and the records are reinserted into the new table.

This strategy is not viable when database utilization is relatively constant with time. Several similar methods, known loosely as *dynamic hashing*, have been devised that allow the table size to grow and shrink dynamically with little overhead [3, 8, 9]. When the load factor gets too high or when buckets overflow, the hash table grows larger and certain buckets are split, thereby reducing the congestion. If the bucket size is rather large, for example, if we allow multiattribute accessing, then coalesced hashing can be used to organize the records within a block, as explained above, thus combining this technique with coalesced hashing in a truly dynamic way.

8. Conclusions

Coalesced hashing is a conceptually elegant and extremely fast method for information storage and retrieval. This paper has examined in detail several practical issues concerning the implementation of the method. The analysis and programming techniques presented here should allow the reader to determine whether coalesced hashing is the method of choice in any given situation, and if so, to implement an efficient version of the algorithm.

The most important issue addressed in this paper is the initialization of the address factor β. The intricate optimization process discussed in Sec. 4 and the Appendix can in principle be applied to any implementation of coalesced hashing. Fortunately, there is no need to undertake such a computational burden for each application, because the results presented in this paper apply to most reasonable implementations. The initialization $\beta \approx 0.86$ is recommended in most cases, because it gives near-optimum search performance for a wide range of load factors. The graph in Fig. 2 makes it possible to fine-tune the choice of β, in case some prior knowledge about the types and frequencies of the searches is available.

The comparisons in Sec. 5 show that the tuned coalesced hashing algorithm outperforms several popular hashing methods when the load factor is greater then 0.6. The differences are more pronounced for large records. The inner search loop in Algorithm C is very short and simple, which is important for practical implementations. Coalesced hashing has the advantage over other chaining methods that it uses only one link field per slot and can achieve full storage utilization. The method is especially suited for applications with a constrained amount of memory or with the requirement that the records cannot be relocated after they are inserted.

In applications where deletions are necessary, one of the strategies described in Sec. 6 should work well in practice. However, research remains to be done in several areas including the analysis of the current deletion algorithms and the design of new strategies that hopefully will preserve randomness. The variant methods in Sec. 7 also pose interesting theoretical and practical open problems. The search performance of varied-insertion coalesced hashing is slightly better than Algorithm C; research is currently underway to analyze its performance and to determine the optimum setting β_{opt}. One exciting aspect of coalesced hashing is that it is an extremely good technique which very likely can be made even more applicable when these open questions are solved.

Appendix

For purposes of average-case analysis, we assume that an unsuccessful search can begin at any of the M address region slots with equal probability. This includes the special case of insertion. Similarly, each record in the table has the same chance of being the object of any given successful search. In other words, all searches and insertions involve *random* keys. This is sometimes called the *Bernoulli probability model.*

The asymptotic formulas in this section apply to a random M'-slot coalesced hash table with address region size $M = \lceil \beta M' \rceil$ and with $N = \lceil \alpha M' \rceil$ inserted records, where the load factor α and the address factor β are constants in the ranges $0 \le \alpha \le 1$ and $0 < \beta \le 1$. Formal derivations are given in [10, 11, 13].

Number of Probes Per Search

The expected number of probes in unsuccessful and successful searches, respectively, as $M' \to \infty$ is

$$C'_N(M', M) \sim \begin{cases} \dfrac{\alpha}{\beta} + e^{-\alpha/\beta} & \text{if } \alpha \le \lambda\beta \\ \dfrac{1}{\beta} + \dfrac{1}{4}\left(e^{2(\alpha/\beta - \lambda)} - 1\right)\left(3 - \dfrac{2}{\beta} + 2\lambda\right) & \\ \quad - \dfrac{1}{2}\left(\dfrac{\alpha}{\beta} - \lambda\right) & \text{if } \alpha \ge \lambda\beta \end{cases} \quad \text{(A1)}$$

$$C_N(M', M) \sim \begin{cases} 1 + \dfrac{1}{2}\dfrac{\alpha}{\beta} & \text{if } \alpha \le \lambda\beta \\ 1 + \dfrac{1}{8}\dfrac{\beta}{\alpha} \cdot & \\ \quad \left(e^{2(\alpha/\beta - \lambda)} - 1 - 2\left(\dfrac{\alpha}{\beta} - \lambda\right)\right) & \\ \quad \left(3 - \dfrac{2}{\beta} + 2\lambda\right) & \\ \quad + \dfrac{1}{4}\left(\dfrac{\alpha}{\beta} + \lambda\right) + \dfrac{1}{4}\lambda & \\ \quad \left(1 - \dfrac{\lambda\beta}{\alpha}\right) & \text{if } \alpha \ge \lambda\beta \end{cases} \quad \text{(A2)}$$

where λ is the unique nonnegative solution to the equation

$$e^{-\lambda} + \lambda = \frac{1}{\beta} \qquad \text{(A3)}$$

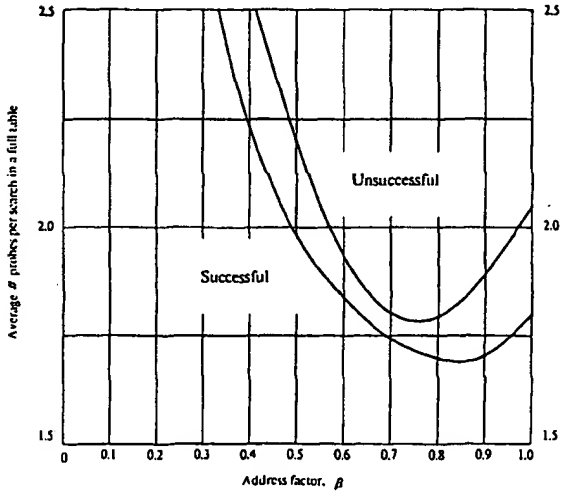
For each address region size M, the average number of probes per search is maximized when the table is full. Figure 10 graphs these maximum values $C'_{M'}(M', M)$ and $C_{M'}(M', M)$ as a function of the address factor β. The choice $\beta \approx 0.782$ yields the best bound on the number of unsuccessful search probes, namely, $C'_{M'}(M', M) \approx 1.79$. If we set $\beta \approx 0.853$, we get the corresponding successful search bound, which is $C_{M'}(M', M) \approx 1.69$.

Although formulas (A1) and (A2) appear a bit formidable at first, they carry a good deal of intuitive meaning. The variable λ is defined to be L/M, where L is the average number of inserted records when the cellar first gets full. The condition $\alpha < \lambda\beta$ means that with high probability the cellar is not yet full. In this case, the chains have not coalesced, so the formulas are identical to those for separate chaining, which are given in [7]. In the other situation, when $\alpha > \lambda\beta$ and the cellar is full with high probability, the formulas are structurally similar to those for the standard coalesced hashing method, given in [7]. At the crossover point $\alpha = \lambda\beta$, both cases of each formula are equal, as can be seen by applying (A3).

The procedure used in Sec. 4 to optimize the number of probes per search for a fixed load factor α makes use of the fact that the optimum address factors β_{opt} which minimize (A1) and (A2) are located somewhere in the "$\alpha \geq \lambda\beta$" region.

In other words, the best address region size M is always large enough so that on the average some amount of coalescing will occur, as we saw in Sec. 1. This can be proved as follows. Inspection of (A3), which relates β and λ, shows that if β increases, then λ decreases, and vice versa. The derivative w.r.t. λ of the expression $\lambda\beta = \lambda/(e^{-\lambda} + \lambda)$ is positive, so β increases when $\lambda\beta$ decreases, and vice versa. This means that the value of β at the cutoff point $\alpha = \lambda\beta$ is the *largest* value of β in the "$\alpha \leq \lambda\beta$" region. (The load factor α is fixed in this analysis.) Since the derivatives w.r.t. β of $C'_N(M', M)$ and $C_N(M', M)$ are both negative when $\alpha \leq \lambda\beta$, a decrease in β would increase $C'_N(M', M)$ and $C_N(M', M)$. Therefore, the minima for the "$\alpha \leq \lambda\beta$" case both occur at the largest possible value for β, namely, the endpoint where $\alpha = \lambda\beta$, which is also part of the "$\alpha \geq \lambda\beta$" region. It also turns out that both formulas (A1) and (A2) are *convex* w.r.t. β, which is also needed for the optimization process in Sec. 4.

Fig. 10. The average number of probes per search used by Algorithm C in a full table, as a function of the address factor β.



MIX Running Times

The MIX running time for unsuccessful searches, which we get by substituting $S = S1 = 0$ into (2), is given by

$$(7C + 4A + 17)u \qquad \text{(A4)}$$

where u is the standard unit of time for MIX computers. The average value of C is $C'_N(M', M)$, which is calculated in (A1). It is convex w.r.t. β and achieves its minimum somewhere in the "$\alpha \geq \lambda\beta$" range. The expected value of a is equal to $1 - E_N/M$, where E_N is defined to be the average number of empty slots in the address region after N records have been inserted. We have

$$E_N \sim \begin{cases} e^{-\alpha/\beta}M & \text{if } \alpha \leq \lambda\beta \\ \dfrac{1-\alpha}{\beta}M & \text{if } \alpha \geq \lambda\beta \end{cases}$$

where λ is defined by (A3). The average value of A achieves its minimum w.r.t. β at the cutoff point $\alpha = \lambda\beta$, where its derivative w.r.t. β is discontinuous. The derivative is negative when $\alpha \leq \lambda\beta$ and positive when $\alpha \geq \lambda\beta$. Since β and $\lambda\beta$ vary in opposite directions, this shows that (A4) is minimized at some point β in the "$\alpha \geq \lambda\beta$" region. (In fact, for $\alpha < 0.5$, the minimum occurs at the endpoint $\alpha = \lambda\beta$.) The optimization in Sec. 4 depends on the empirically verified fact that for each fixed value of α, formula (A4) is *well-behaved.* By that we mean that (A4) is minimized at a unique β_{opt}, which occurs either at the endpoint $\alpha = \lambda\beta$ or at the unique point in the "$\alpha \geq \lambda\beta$" region where the derivative is 0.

The value of A is always 1 in successful searches, since only occupied slots are probed. Substituting $A = 1$ and $S = 1$ into (2), we obtain the following formula for the MIX successful search time:

$$(7C + 18 + 2S1)u \qquad \text{(A5)}$$

Here the averge value of C is $C_N(M', M)$, which is given in (A2). It is convex w.r.t. β and is minimized at some point in the "$\alpha \geq \lambda\beta$" range. The expected value of $S1$ is the average number of records that do *not* collide when inserted, divided by N. That is equal to

$$S1_N = \frac{1}{N} \sum_{0 \le n < N} \frac{E_n}{M}$$

$$\sim \begin{cases} \dfrac{\beta}{\alpha}(1 - e^{-\alpha/\beta}) & \text{if } \alpha \le \lambda\beta \\ \dfrac{1}{\beta}\left(1 - \dfrac{1}{2}\alpha\right) - \dfrac{1}{\alpha}(1 - \beta) & \\ \quad \cdot(1 + \lambda) + \dfrac{1}{2}\dfrac{\lambda^2\beta}{\alpha} & \text{if } \alpha \ge \lambda\beta \end{cases}$$

where λ is defined by (A3). The expected value of $S1$ attains its *maximum* in the "$\alpha \ge \lambda\beta$" region. This gives us no clue as to whether (A5) is well-behaved in the above sense, which is assumed by the optimization procedure, but numerical study verifies that it is true.

Addendum: Since the time this article was submitted, one of the open problems mentioned in Sec. 7 has been solved by Wen-Chin Chen and the author. The analyses of the search times of early-insertion and varied-insertion coalesced hashing appear in "Analysis of Some New Variants of Coalesced Hashing," Department of Computer Science, Brown University, Technical Report No. CS-82-18, June 1982.

Acknowledgments. The author wishes to thank Don Knuth for this encouragement and guidance. Thanks also go to John Savage, whose comments helped improve the organization of this paper, and to Wen-Chin Chen for his help in making some of the diagrams. Many of the calculations were done with the help of the MACSYMA system (which is supported in part by the United States Energy Research and Development Administration and by the National Aeronautics and Space Administration). The submitted version of this article was typeset using the TEX facility at the American Mathematical Society in Providence. Finally, the author would like to acknowledge his family for playing a key role in the hash table examples.

Received 7/81; revised 3/82; accepted 4/82

References

1. Chen, W.C. and Vitter, J.S. Analysis of early-insertion standard coalesced hashing. Department of Computer Science, Brown University, Technical Report CS-82-14, March 1982. Also to appear in *SIAM Journal on Computing*.
2. Carter, J.L. and Wegman, M.N. Universal classes of hash functions. *Journal of Computer and System Sciences 18*, 2 (April 1979), 143–154.
3. Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H.R. Extendible hashing—A fast access method for dynamic files. *ACM Transactions on Database Systems 4*, 3 (Sept. 1979), 315–344.
4. Guibas, L.J. *The Analysis of Hashing Algorithms.* PhD dissertation, Computer Science Department, Stanford University, Technical Report STAN-CS-76-556, August 1976.
5. Knott, G.D. Direct chaining with coalescing lists. To appear in *Journal of Algorithms.*
6. Knuth, D.E. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.* 2nd Ed. Addison-Wesley, Reading, Massachusetts, 1973.
7. Knuth, D.E. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 1973.
8. Larson, P. Linear hashing with partial expansions. Departments of Information Processing and Mathematics, Swedish University of Åbo, Technical Report Ser. A, No. 7, March 1980.
9. Litwin, W. Virtual hashing: A dynamically changing hashing. *Proceedings of 4th Conference on Very Large Databases.* Berlin 1978, 517–523.
10. Vitter, J.S. *Analysis of Coalesced Hashing.* PhD dissertion, Department of Computer Science, Stanford University, Technical Report STAN-CS-80-817, August 1980.
11. Vitter, J.S. Tuning the coalesced hashing method to obtain optimum performance. *Proceedings of 21st Annual Symposium on Foundations of Computer Science.* Syracuse, NY, October 1980, 238–247.
12. Vitter, J.S. A shared-memory implementation for coalesced hashing. *Information Processing Letters 13*, 2 (Nov. 13, 1981), 77–79.
13. Vitter, J.S. Analysis of the search performance of coalesced hashing. (To appear in Journal of the ACM.) An earlier version appeared in Department of Computer Science, Brown University, Technical Report No. CS-71, July 1981.
14. Vitter, J.S. Deletion algorithms for hashing that preserve randomness. *Journal of Algorithms 3* (Sept, 1982) 261–275.
15. Wiederhold, G. *Database Design.* McGraw-Hill, New York, 1977.
16. Williams, F.A. Handling identifiers as internal symbols in language processors. *Comm ACM 2*, 6 (June 1959), 21–24.

Optimal Arrangement of Keys in a Hash Table

RONALD L. RIVEST

Massachusetts Institute of Technology, Cambridge, Massachusetts

ABSTRACT When open addressing is used to resolve collisions in a hash table, a given set of keys may be arranged in many ways, typically this depends on the order in which the keys are inserted It is shown that arrangements minimizing either the average or worst-case number of probes required to retrieve any key in the table can be found using an algorithm for the assignment problem. The worst-case retrieval time can be reduced to $O(\log_2(M))$ with probability $1 - \epsilon(M)$ when storing M keys in a table of size M, where $\epsilon(M) \to 0$ as $M \to \infty$ We also examine insertion algorithms to see how to apply these ideas for a dynamically changing set of keys

KEY WORDS AND PHRASES hashing, collision resolution, searching, assignment problem, optimal algorithms, database organization

CR CATEGORIES 3 74, 5 41

"Spread the table and contention will cease " Old English proverb [11, #272 6]

1. *Introduction*

We consider schemes to optimize the placement of keys in a hash table when open addressing is used to resolve collisions. More precisely, we begin with the observation that a given set of keys may be inserted into a hash table in many different orders, yielding arrangements of the keys in the table of varying efficiency. Typically, the user has no control over the order in which the keys are inserted; he must accept them in the order in which they arrive. However, the previous observation that there exist many different arrangements of the given set of keys raises the following questions:

(1) How can one determine that arrangement which minimizes either the average or worst-case number of probes to retrieve a key in the table? In Section 2 we show that this problem is an instance of the well-known "assignment problem," for which efficient algorithms exist.

(2) What is the expected value of the worst-case number of probes required to retrieve a key from a full table that has been optimally arranged using the assignment algorithm? In Section 3 it is proved that this value is $O(\log_2(M))$ for a table of size M containing M keys. The proof is modeled on a result by Erdös and Renyi [2] concerning the permanent of a random matrix. This result demonstrates that we can use hashing to achieve "good" (i.e. $O(\log_2(M))$) worst-case performance if we take the time to optimize the arrangement of the keys in the table. Traditionally hashing has

This research was prepared with the support of the National Science Foundation under Research Grant GJ-43534X, Contract DCR74-12997, and Research Grant MCS76-14294

Author's address Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139

been viewed as excellent on the average, but horrible in the worst case. We see therefore that this need not be so.

(3) The results mentioned above require that an $M \times M$ assignment problem be solved to optimize the placement of M keys in a table of size M. A natural question to ask is, "Is it possible to solve the assignment problem efficiently 'incrementally,' so that the new keys can be added to the table in such a way that the optimality of the overall arrangement is maintained?" In Section 4 this problem is studied and it is shown that for table densities less than approximately 0.415, it is possible to insert a key and maintain overall optimality by solving an assignment problem no larger than 10×10, whereas for larger densities the entire $M \times M$ assignment problem must apparently be solved.

Overall, we view the contribution of this paper to be the introduction of the assignment algorithm for the placement of keys in a hash table, and the demonstration that efficient worst-case retrieval can be achieved thereby, even in a full table.

We proceed now to define our terminology and to introduce the "standard" algorithm for inserting a key into a hash table. Let $\mathcal{K} = \{K_1, K_2, \ldots, K_N\}$ be a set of N keys, and let an array T_i for $1 \leq i \leq M$ be a set of M memory locations (the hash table) which will be used to store $\mathcal{K}$. Each table position may hold either a single key or the special symbol *empty*. We assume $N \leq M$. When open addressing is used to resolve collisions a "hashing function" $h: U \times \{1, 2, \ldots, M\} \rightarrow \{1, 2, \ldots, M\}$ is used, mapping the set U of all possible keys (that is, $\mathcal{K}$ may be any N-subset of U) and probe numbers into the set of memory locations. We assume for any key $K \in U$ that the sequence $h(K, 1), h(K, 2), \ldots, h(K, M)$ is a permutation of $\{1, 2, \ldots, M\}$ To store the key K in the table using the standard insertion algorithm the locations $T_{h(K,1)}, T_{h(K,2)}, \ldots$ are successively examined until an empty location is found or until K is found already present in the table. The following program makes this precise.

THE "STANDARD" INSERTION ALGORITHM

Input: A key K, a hash table T, a hash function h
Output: None. T is modified to contain K, unless K is already present
Procedure

```
j := 0,
repeat j := j + 1,
       i := h(K, j),
       if T_i = empty then T_i := K
until  T_i = K,
```

Note that T must contain at least one empty location if K is not already in the table, if the loop is to terminate properly. The value of j at termination, which is the number of probes required to insert K, is taken to be the cost of inserting K.

A similar procedure searches for the presence of a key K in T (replace the assignment statement "$T_i := K$" by "return (K not present)") If the **repeat** loop terminates normally then T_i contains the previously stored key K. The value of j at termination is taken to be the cost of searching for K.

Knuth [6] studies hashing algorithms in detail, giving alternative methods for handling "collisions" (the case when $h(K_i, 1) = h(K_j, 1)$ for $K_i \neq K_j$) and several open-addressing hash functions h. The reader who is unfamiliar with hashing algorithms should find it profitable to consult his text.

2 *Optimal Arrangements*

In this section we give precise definitions of when an arrangement minimizes the average or worst-case retrieval time, and then show that there always exists some ordering such that if the keys had been inserted by the standard algorithm in that order, the optimal arrangement results. Then it is shown that the assignment algorithm

can be used to arrange the keys so as to minimize either the average or worst-case retrieval time.

The arrangement of the keys $\mathcal{K}$ in the hash table depends on the order in which they were inserted, if the standard insertion algorithm is used. For example, let U be the set of natural numbers and let $h(K, j)$ be the jth decimal digit of K. Inserting the set $\mathcal{K} = \{1423, 1234, 3412, 2341\}$ into an empty table in that order results in the arrangement α:

Location:	1	2	3	4
Contents:	1423	1234	3412	2341

whereas inserting them in the order 1234, 2341, 1423, 3412 results in α':

Location:	1	2	3	4
Contents	1234	2341	3412	1423

Let $\alpha: \mathcal{K} \to \{1, 2, \ldots, M\}$ be called an *arrangement*; $\alpha(K_i) = j$ means that $T_j = K_i$. Of course α must be one-to-one. Let $A(\mathcal{K}, M)$ denote the set of all arrangements of $\mathcal{K}$ in $T_1, \ldots, T_M$.

Let $p(K, \alpha)$ denote the number of probes required to retrieve a key K under arrangement α; the average $\text{avg}(\alpha) = (1/N) \sum_{K \in \mathcal{K}} p(K, \alpha)$ and worst-case $\text{wc}(\alpha) = \max\{p(K, \alpha) | K \in \mathcal{K}\}$ number of probes to retrieve any key in T are then definable. We have $\text{avg}(\alpha) = 7/4$, $\text{wc}(\alpha) = 3$, $\text{avg}(\alpha') = 5/4$, and $\text{wc}(\alpha') = 2$ in the above examples.

Define an arrangement $\alpha \in A(\mathcal{K}, M)$ to be *valid* if all the positions $h(K, 1), h(K, 2), \ldots, h(K, p(K, \alpha) - 1)$ are nonempty for every key K in $\mathcal{K}$. An arrangement is valid iff every key K in $\mathcal{K}$ is retrievable using the search algorithm of Section 1 Similarly define an arrangement to be *feasible* if it is the result of inserting the keys in $\mathcal{K}$ into an empty table sequentially in some order; necessarily every feasible arrangement is valid.

Valid arrangements which are not feasible are possible; consider the following arrangement using the hash function h from our previous example:

Location:	1	2	3	4
Contents:	empty	empty	4321	3412

The number of feasible arrangements depends on $\mathcal{K}$ and h. It is no larger than $N!$ (the number of ways to enter the keys), but may be as low as 1 if no collisions occur. Similarly the number of valid arrangements can vary between 1 and $N!$. For example, only one valid arrangement exists if no collisions occur and $h(K_i, 1) \neq h(K_j, 2)$ for all K_i, K_j in $\mathcal{K}$. The upper bound of $N!$ on the number of valid arrangements is obtained by induction on N, using the fact that $p(K, \alpha) \leq N$ for any valid arrangement and all keys $K \in \mathcal{K}$. We may store K_N in any of N positions $h(K_N, i)$ for $1 \leq i \leq N$; if we then delete K_N from $\mathcal{K}$ and $h(K_N, i)$ from the probe sequence $h(K_j, 1), \ldots, h(K_j, M)$ for every $j < N$ we see that every valid arrangement of $\mathcal{K}$ induces a valid arrangement of $\mathcal{K}$-$\{K_N\}$ in locations $\{j | 1 \leq j \leq M \text{ and } j \neq h(K_N, i)\}$ using the modified probe sequences.

We define an arrangement $\alpha(\mathcal{K}, M)$ to be *optimal* if either $\text{avg}(\alpha)$ or $\text{wc}(\alpha)$ is minimal over all arrangements in $A(\mathcal{K}, M)$; the terms average-optimal and worst-case-optimal will distinguish these cases.

PROPOSITION 1. *A feasible optimal arrangement always exists.*

PROOF If a minimal arrangement α is not feasible, then there exists a set $\{K_{i_0}, K_{i_1}, \ldots, K_{i_{r-1}}\}$ of keys, none of which can be entered first since they form a "blocking cycle": There is a set of integers t_j for $0 \leq j \leq r - 1$ such that $h(K_{i_j}, p(K_{i_j}, \alpha)) = h(K_{i_{(j+1) \bmod r}}, t_{(j+1) \bmod r})$ and $t_j < p(K_{i_j}, \alpha)$ for $0 \leq j \leq r - 1$. But clearly $p(K_{i_j}, \alpha)$ can be reduced by setting $\alpha(K_{i_j})$ to $h(K_{i_j}, t_j)$ for $0 \leq j \leq r - 1$. Since $\text{avg}(\alpha)$ strictly decreases, a feasible optimal arrangement can always be found after a finite number of blocking cycles have been removed in this fashion. □

Proposition 1 suggests an algorithm for finding optimal arrangements: enumerating all feasible arrangements; however, better methods exist.

PROPOSITION 2 *Optimal arrangements can be found by using an algorithm for the assignment problem*

PROOF. The assignment problem [7] can be stated as follows.

Let N and M be given, with $N \leq M$, and let $\{a_{ij} | 1 \leq i \leq N, 1 \leq j \leq M\}$ be a matrix of nonnegative real numbers. The classic example specifies for each of M men and N jobs, the "inefficiency" a_{ij} of man j in job i. The objective is to find an assignment $i \rightarrow \alpha(i)$ of jobs to men such that the sum $\sum_{1 \leq i \leq N} a_{i,\alpha(i)}$ is minimized, subject to the constraint that no man is assigned to more than one job.

We can apply this directly to the problem of finding average-optimal arrangements by letting a_{ij} be the integer such that $h(K_i, a_{ij}) = j$, denoting the cost of assigning K_i to T_j. The average number of probes required to retrieve a key in the optimized table is then just the total "inefficiency" divided by N We observe that if the various keys have associated retrieval probabilities, then the arrangement that minimizes the expected retrieval cost can be found in the same manner; we need only multiply each a_{ij} by the probability that K_i will be retrieved.

Similarly, we can minimize the worst-case cost by choosing a_{ij} to be N^l, where l is the integer such that $h(K_i, l) = j$. Since the key with highest cost determines the order of the total cost, minimizing the total cost here minimizes the worst-case cost. □

Having observed that our problem can be formulated as an instance of the assignment problem, it is of interest to know how quickly a solution can be determined. The general $N \times M$ assignment problem can be solved in time $O(NM^2)$ [8]; the space required is $O(N + M)$ if the matrix entries a_{ij} can be computed in constant time from K_i, h, and j. When all the matrix entries are small integers (as when we are finding the average-optimal arrangement), it may be possible to improve this time bound somewhat, but the author was unable to find a more efficient procedure.

Worst-case optimal arrangements can be determined in time $O(BM(M, N) \cdot \log_2(N))$, where $BM(M, N)$ is the time required to solve an $M \times N$ bipartite matching problem. The procedure, pointed out to the author by Vuillemin, is to use binary search on the worst-case cost: It is possible to test if the optimal worst-case cost is less than or equal to a given value w by solving the corresponding maximal matching problem. The graph used has N vertices x_i, M vertices y_j, and an edge (x_i, y_j) iff $a_{ij} \leq w$. Intuitively, there is an edge from x_i to y_j if and only if table position T_j is one of the first w positions in the probe sequence for K_i. There will be a matching of size N in this graph if and only if there is an arrangement of the keys in the table such that every key can be retrieved with no more than w probes. Since $BM(M, M) = O(M^{2.5})$, we obtain an $O(M^{2.5} \log(M))$ algorithm for the case $N = M$.

3 *Efficiency of the Worst-Case Optimal Arrangements*

In this section we prove that even if the hash table is full ($N = M$), we can expect the worst-case optimal arrangement to have a worst-case cost of $O(\log(M))$ with a probability approaching one very rapidly as $M \rightarrow \infty$. Although a worst-case cost of $O(\log(M))$ can obviously not be guaranteed (since there is a finite chance that all keys have the same probe sequence, for example), the odds are overwhelming that with a random hash function and a random set of keys, there is some arrangement of those keys yielding a worst-case cost of $O(\log(M))$. This compares favorably with standard techniques such as binary search trees which also require $O(\log_2(N))$ time to retrieve a key, especially in situations where the set of keys is static (since updating an optimized hash table can be expensive).

The proof is modeled very closely after a similar result of Erdos and Renyi [2], who show that a random $n \times n$ matrix of 0's and 1's containing $N(n)$ 1's has a nonzero permanent with probability approaching 1 as $n \rightarrow \infty$ if $\lim_{n \rightarrow \infty}(N(n) - \log(n))/n = \infty$. The permanent of an $n \times n$ matrix $\{a_{ij}\}$ is defined to be $\sum a_{1i_1} a_{2i_2} \cdots a_{ni_n}$, where the summation is over all permutations $(i_1, \ldots, i_n)$ of $\{1, \ldots, n\}$. The permanent of a 0-1

matrix $\{a_{ij}\}$ is the number of matchings of size n in a bipartite graph whose adjacency matrix is $\{a_{ij}\}$ Ryser [10] discusses the permanent in some detail

Let $\mathcal{M}(M, N, w)$ denote the set of all 0-1 matrices with M columns, N rows, and exactly w 1's per row. Obviously $|\mathcal{M}(M, N, w)| = \binom{M}{w}^N$. We say a matrix $\{m_{ij}\} \in \mathcal{M}(M, N, w)$ contains N independent 1's iff there exists a function $\alpha:\{1, \ldots, N\} \to \{1, \ldots, M\}$ such that $\alpha(i) \neq \alpha(j)$ for $i \neq j$ and $m_{i,\alpha(i)} = 1$ for $1 \leq i \leq N$. Let $P(M, N, w)$ denote the probability that a matrix in $\mathcal{M}(M, N, w)$ contains N independent ones.

The interpretation to matrices of $\mathcal{M}(M, N, w)$ is as follows. Each such matrix has N rows (corresponding to a set of N keys) and M columns (one for each position in the hash table) Position i, j will be a 1 *iff* key i can be stored in position j with a retrieval cost of w or less. Therefore each row has exactly w 1's Such a matrix is the adjacency matrix of one of the bipartite graphs described in the last paragraph of Section 2 A matrix in $\mathcal{M}(M, N, w)$ will have N independent ones iff its corresponding bipartite graph has a matching of size N. This will happen iff there exists an arrangement of the keys so that every one can be retrieved with w probes or less.

We identify $P(M, N, w)$ with the probability that a random set of N keys can be arranged in a hash table of size M so that the worst-case retrieval cost is at most w. This will be accurate if every set of w locations is equally likely to be the set of w locations first probed for a random key k This will happen, for example, if every permutation of $\{1, . , M\}$ is equally likely to be a probe sequence. Each matrix in $\mathcal{M}(M, N, w)$ then corresponds in a natural fashion to the characteristic matrix describing, for a random set of N keys, which locations are usable if the worst-case cost is constrained to be at most w The existence of N independent 1's corresponds to the existence of an arrangement with worst-case cost of at most w; and by Proposition 1 the existence of a feasible, valid arrangement with worst-case cost at most w is thereby implied.

We have $P(M, N, w) \geq P(M, M, w)$ for $1 \leq N \leq M$ since the first N rows of a matrix in $\mathcal{M}(M, M, w)$ which contains M independent 1's must contain N independent 1's. We therefore proceed to show the following.

PROPOSITION 3. $\lim_{M\to\infty} P(M, M, 4\log(M)) = 1$.

PROOF. This result says that we can expect to find an arrangement of M keys in a table of size M such that no key requires more than 4 $\log(M)$ probes to be retrieved. By the theorems of Frobenius [3] and König [7], $1 - P(M, M, w)$ is equal to the probability that a matrix in $\mathcal{M}(M, M, w)$ has k rows (or columns) and $M - k - 1$ columns (or rows) that contain all the 1's, for some k, $0 \leq k \leq M - 1$. (The result of Frobenius and König says that in an $M \times M$ matrix of 0's and 1's the minimal number of lines (i.e. rows or columns) which contain all the 1's is equal to the size of the maximum set of 1's which can be found which are pairwise independent (no two in the same line).) Thus $1 - P(M, M, w)$ is the probability that there are $M - 1$ or fewer lines which contain all the 1's.

Let $Q_k(M, N, w)$ denote the probability that a matrix in $\mathcal{M}(M, N, w)$ has k rows (or columns) and $N - k - 1$ columns (or rows) containing all the 1's, and k is the least such number for $0 \leq k \leq M/2$. Then

$$1 - P(M, N, w) = \sum_{k=0}^{\lfloor M/2 \rfloor} Q_k(M, N, w).$$

We show that for all k, $0 \leq k \leq \lfloor m/2 \rfloor$, if $w \geq 4 \log_2(M)$ then $Q_k(M, M, w) \to 0$. To do this we divide Q_k into two parts,

$$Q_k(M, M, w) = f_k(M, M, w) + g_k(M, M, w),$$

where f_k is the probability that k rows and $M - k - 1$ columns cover all the 1's and g_k is the probability that k columns and $M - k - 1$ rows cover all the 1's (k is each case being minimal).

Case 1. k rows and $M - k - 1$ columns contain all the 1's, for some $k \leq M/2$.

Those matrices in $\mathcal{M}(M, M, w)$ having a minimal number k of rows and $M - k - 1$ columns containing all the 1's can be displayed as in Figure 1, after an appropriate permutation of the rows and columns Each row of submatrix B must contain two 1's under our assumption that k is minimal (if not, we could include the column, and exclude the row, of the 1 in matrix B which is in a row of B containing no other 1's). The fraction $f_k(M, M, w)$ of matrices of this type is less than

$$\left(\binom{M}{k}\binom{M}{k+1}\binom{M-k-1}{w}^{M-k}\left(\binom{M}{w} - \binom{M-k-1}{w} - (k+1)\binom{M-k-1}{w-1}\right)^{k}\right)\Big/\binom{M}{w}^{M},$$

whose logarithm is bounded above by

$$[(2k+1) - w(M-k)]\log(M) + w(M-k)\log(M-k-1) - k\log(k) - (k+1)\log(k+1) \le (2k+1)\log(M) - w(k+1)/2.$$

Thus if $w \ge 4\log(M)$, $Q_k(M, M, w) \to 0$ as $M \to \infty$.

Case 2 k columns and $M - k - 1$ rows contain all the 1's, for some $k \le M/2$ (Figure 2).

The fraction $g_k(M, M, w)$ of matrices of this type is less than

$$\binom{M}{k}\binom{M}{k+1}\binom{M}{w}^{-(k+1)}\binom{k}{w}^{k+1},$$

whose logarithm is bounded above by

$$(2k+1)\log(M) - w(k+1)\log(w),$$

so that $g_k(M, M, w) \to 0$ with M if $w = 2\log(M)$. Since $Q_k(M, M, w) = f_k(M, M, w) + g_k(M, M, w)$, we are finished with the proof. □

This result says that in a full table arranged so as to minimize the worst-case retrieval time, the worst-case retrieval time should be $O(\log(M))$. This follows from Proposition 3 since the existence of a set of M independent 1's in a matrix in $P(M, M, w)$ corresponds to an arrangement of M keys in a table of size M with worst-case retrieval time no more than w. This result is the best possible (up to a constant multiplicative factor) due to a result of Gonnet [4]: The worst-case retrieval time must be at least $\ln(M) + O(1)$.

A study of the related question of the expected value of the average number of probes required to retrieve a key in a full table which is average-optimal is given in [5]. (Less than two probes per key are required.)

4. *Insertion Algorithms Which Maintain Optimality*

We now turn our attention to the problem of maintaining the optimality of an arrangement as new keys are inserted into a table. The main result of this section is that if the table is not too densely filled, then a new key can be inserted into the table

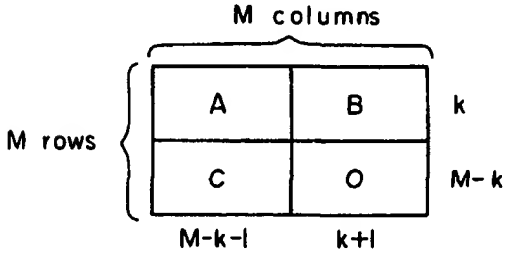


FIG 1

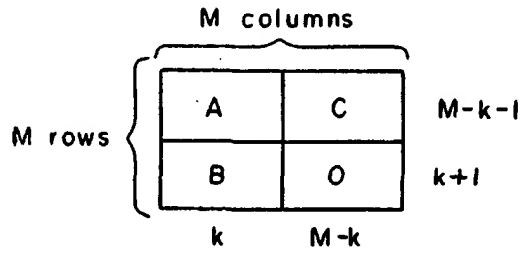


FIG 2

and the new optimal arrangement computed by solving a small (e.g 10×10) assignment problem. This result is obtained by a rather complicated analysis using generating functions.

We first examine an insertion algorithm due to Brent [1] and demonstrate that it does not maintain optimality. Of course, Brent only intended his algorithm to be a good heuristic, a means of inserting each new key in such a fashion that the increase in average retrieval cost is kept reasonably low

Brent's algorithm works as follows. Let K denote the new key being inserted, and suppose positions $h(K, 1), \ldots, h(K, s)$ are already occupied with keys $K_1, K_2, \ldots, K_s$, and that $T_{h(K,s+1)}$ is empty. Let r_i denote the number of probes required to retrieve K_i, so that $h(K_i, r_i) = h(K, i)$. Furthermore, let s_i denote $\min\{j | T_{h(K_i,j)} = \textit{empty}\}$, the number of probes required to retrieve K_i if we move it to position $h(K_i, s_i)$. Then $(i + (s_i - r_i))/(N + 1)$ is the increase in the average retrieval cost caused by moving K_i to position $h(K_i, s_i)$ and storing K in position $h(K, i)$. Brent chooses between storing K in position $h(K, s + 1)$ and moving that K which minimizes $i + (s_i - r_i)$ by comparing $(s + 1)$ to $\min_i\{i + s_i - r_i\}$.

In fact, the following example demonstrates that no algorithm which only moves keys forward in their probe sequence (that is, moves K from $h(K, i)$ to $h(K, i')$ for $i' > i$) can always arrive at the optimal arrangement. Consider the following arrangement (using the hash function of our previous examples), which is both average and worst-case optimal:

Location	1	2	3	4	5	6	7
Contents	1273456	1234567	3456712	4567123	5671234	6712345	empty

If the key 2345671 is now inserted, the only way to maintain optimality is to move 1273456 to location 7, move 1234567 (backward) to position 1, and then store 2345671 in position 2

Since Brent's algorithm is the only published algorithm which moves previously inserted keys when inserting a new key, we see that no existing insertion algorithm can maintain optimality for arbitrary hash functions. It is interesting to note, however, that for certain open-addressing collision-resolution schemes the standard insertion algorithm maintains average-optimality. We say that a hash function h exhibits *primary clustering* if $h(K_i, j) = h(K_{i'}, j')$ implies that $h(K_i, j + l) = h(K_{i'}, j' + l)$ for $0 \le l \le M - \min(j, j')$ for any $K_i, K_{i'}$. Linear probing ($h(K, i) \equiv h(K, 1) + (i - 1)$, mod M) is perhaps the best-known example of a collision-resolution scheme exhibiting primary clustering, and all primary clustering schemes are in fact isomorphic to linear probing in a natural manner

PROPOSITION 4. *If h exhibits primary clustering, then the usual insertion algorithm maintains average-optimality*

PROOF. This theorem is due to Peterson [9]; the proof is also given in Knuth [6, p. 531]. Knuth also remarks that if the keys have associated retrieval probabilities, then the average-optimal arrangement can be achieved by using the standard insertion routine to insert the keys one by one into the table, in order of decreasing request probabilities. □

In spite of the fact that for linear probing the standard insertion algorithm maintains average-optimality, other hashing schemes are to be preferred, since the expected retrieval cost in the average-optimal scheme for a primary-clustering hashing function generally exceeds the expected cost for other schemes, even if average-optimality is not maintained.

We now turn our attention to the task of finding an insertion algorithm that will maintain the optimality of an arrangement. In essence, we need an algorithm to solve the assignment problem "incrementally."

One approach is to observe that if N/M is small enough (how small this is we shall determine), then the number of keys already in the table which we need to consider moving might be reasonably small. Brent considers moving only those keys on the

probe sequence of the new key K; if we also consider moving all of the keys on their probe sequences, and so on, we can determine the maximum set $\mathcal{S}$ of keys that might need to be moved. Similarly we let $\mathcal{T}$ denote the set of locations that $\mathcal{S}$ might occupy in the optimized table; it suffices then to solve the assignment problem for placing $\mathcal{S}$ into $\mathcal{T}$, rather than $\mathcal{K} \cup \{K\}$ into T.

Define, for a given arrangement α, the functions:

$$\pi(K) = \min\{j \mid h(K, j) = \text{empty}\},$$
$$\sigma(K) = \{K_i \mid \alpha(K_i) = h(K, j) \text{ for some } j < \pi(K)\},$$
$$\tau(K) = \{i \mid h(K, j) = i \text{ for some } j \leq \pi(K)\}.$$

Then

$$\mathcal{S}(K) = \{K\} \cup \{\mathcal{S}(K_i) \mid K_i \in \sigma(K)\},$$
$$\mathcal{T}(K) = \tau(K) \cup \{\mathcal{T}(K_i) \mid K_i \in \sigma(K)\}$$

define by means of their minimal solutions the sets $\mathcal{S}$ and $\mathcal{T}$ of keys and positions relevant to the insertion of K into an arrangement α

Let $\beta = N/M$ denote the "loading factor" of the existing arrangement α. In order to estimate the expected size $\mathcal{S}(K)$, we assume that the hashing function is uniform in the sense that every permutation of $\{1, \ldots, M\}$ is equally likely to be a probe sequence of some key K. We can then use the approximation $\text{Prob}(\pi(K) = i) = (1 - \beta)\beta^{i-1}$

Let s_i denote the probability that $|\mathcal{S}(K)| = i$, and let

$$S(z) = \sum_{i=1}^{\infty} s_i z^i$$

denote the corresponding generating function. We shall develop an equation for $S(z)$ which depends on the generating function:

$$P(z) = \sum_{i=1}^{\infty} p_i z^i$$

(where p_i is the probability that, for a key K' already stored in T, $\alpha(K') = h(K', i)$). However, determining $P(z)$ for optimized hash tables remains an open problem, so we shall approximate $S(z)$ after we develop the correct defining equation.

Let $C(z) = \sum_{i=1}^{\infty} c_i z^i$ be the generating function with coefficients c_i equal to the probability that the "contribution" of a key K' on the probe sequence of the new key K to $S(K)$ is i keys. Therefore

$$S(z) = \sum_{i=0}^{\infty} (1 - \beta)\beta^i [C(z)]^i \cdot z,$$

since there is a probability of $(1 - \beta)\beta^i$ that $\pi(K) = i + 1$ (that is, there are i keys on the probe sequence for the new key K). The final z is for the key K itself.

Similarly we can define

$$C(z) = \left[\sum_{i=1}^{\infty} p_i (C(z))^{i-1}\right] \cdot \left[\sum_{i=0}^{\infty} (1 - \beta)\beta^i (C(z))^i\right] \cdot z$$

(or equivalently,

$$(1 - \beta C(z)) \cdot (C(z))^2 = (1 - \beta) P(C(z)) z).$$

The first term accumulates the contributions of those keys K'' on the probe sequences of a key K' on the probe sequence for K, such that K'' occurs before K' in the probe sequence for K' The second term adjusts for those keys K'' occurring after K' in the probe sequence for K'. Finally, the third term z is for the key K' itself.

The expected size of $\mathcal{S}(K)$ is $S'(1)$; and

$$S'(z) = \frac{d}{dz}\left(\frac{(1 - \beta)z}{(1 - \beta C(z))}\right) = \frac{(1 - \beta C(z))(1 - \beta) + (1 - \beta) z \beta C'(z)}{(1 - \beta C(z))^2}$$

so that

$$S'(1) = 1 + \frac{\beta C'(1)}{(1 - \beta)}.$$

Now

$$(1 - \beta C(z))2C(z)C'(z) - \beta C'(z)(C(z))^2 = (1 - \beta)[P'(z)C'(z)z + P(C(z))]$$

so we obtain

$$C'(1) = (1 - \beta)/(2 - 3\beta - (1 - \beta)P'(1))$$

and thus

$$S'(1) = 1 + \beta/(2 - 3\beta - (1 - \beta)P'(1)).$$

Unfortunately, $P(z)$ is unknown. We observe, however, that $S'(1)$ can be expected to remain finite as long as $P'(1) \leq (2 - 3\beta)/(1 - \beta)$. Since $P'(1)$ is the expected number of probes required to retrieve a key from an optimized table, it is bounded above by the expected number of probes required to retrieve a key from a table organized with any open-addressing hashing method. For uniform probing (all probes sequences equally likely) we have [6]

$$P'(1) \cong \beta^{-1}\log(1/(1 - \beta))$$

approximately. Substituting this into the final equation for $S'(1)$ yields Figure 3; we see that the size of the relevant assignment problem is reasonably small (say 10 keys or less) as long as $\beta \leq 0.4$ roughly The function $S'(1)$ has a pole $\beta = 0.41466541$; for loading densities less than this we can expect the number of relevant keys to be finite. In practice we should expect to be able to handle even higher loading densities without much trouble, since our formulas for S, C, and P explicitly ignore the probability of overlapping probe sequences Furthermore, replacing $P(z)$ by its correct definition (rather than the one for uniform probing) should yield a definite improvement.

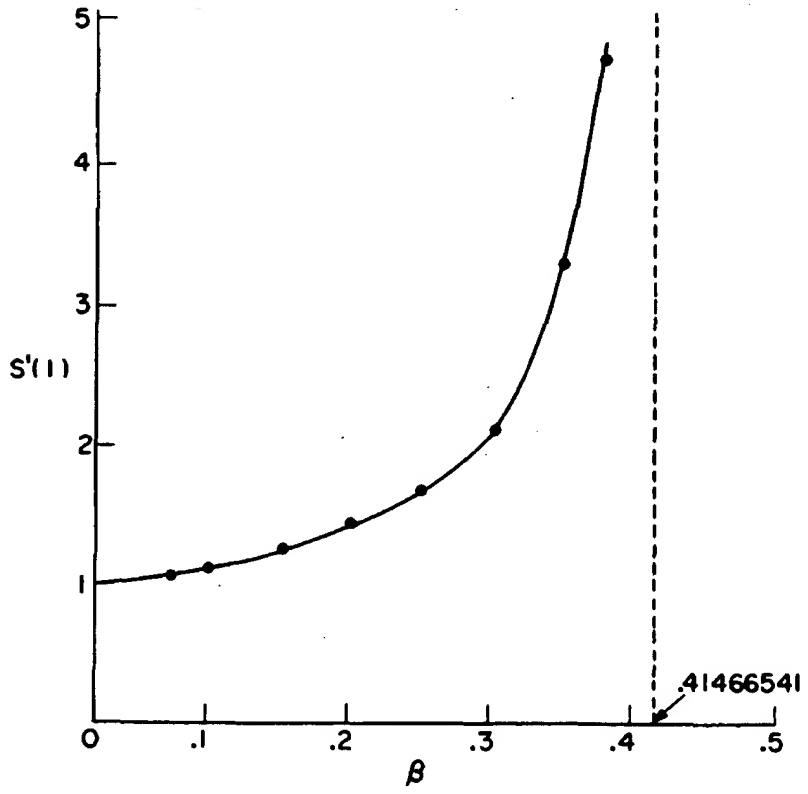


FIG 3

The result of this rather complicated analysis is that if the loading density of the file is less than roughly 0.4 we can hope to insert a new key K into the table by solving a small assignment problem. For higher densities the problem is inherently a global one apparently; we must consider for relocation a considerable number of keys.

5. *Discussion and Conclusions*

In this paper we have shown how to arrange a set of keys in a hash table so as to minimize the expected (or worst-case) number of probes required to retrieve a key. Our analysis demonstrates that the worst-case cost can be reduced to $O(\log_2(M))$ in almost all cases. (In practice it should be possible to achieve $O(\log_2(M))$ in all cases with very little work, since a set of keys which has an optimized cost that is too large can, by choosing another hash function randomly, be expected to yield an $O(\log_2(M))$ cost.)

Our analysis assumes that uniform hashing is used, however; an open problem is to confirm this result for the more common techniques such as double hashing.

We have also examined briefly a technique for inserting a new key into an optimized table so as to maintain optimality of the arrangement. Our result here is that as long as the loading factor is less than 0.41 (approximately), we can usually insert a new key and maintain optimality by solving a small (approximately 10-element) assignment problem. For tables of higher density one must apparently solve an assignment problem which involves most of the keys previously stored. (By saving the primal and dual variables of the previous solution, one can significantly speed up the solution of the new problem, but the extra storage required might better be used to store the keys themselves, thereby reducing the overall density.)

The reader is encouraged to consult the excellent article by Gonnet and Munro [5], which gives explicit listings of algorithms for optimizing the arrangement of keys in a hash table and tight results on the expected number of probes required to retrieve a key from an average-optimal table.

The techniques described here should be most useful when the hash table is relatively static, with the number of retrievals considerably exceeding the number of insertions Large databases are often of exactly this nature, and frequently utilize hashing techniques

ACKNOWLEDGMENT. I would like to thank Professor Donald Knuth for suggesting directions in which to extend a previous draft of this paper.

REFERENCES

1 BRENT, R P Reducing the retrieval time of scatter storage techniques *Comm ACM 16*, 2 (Feb 1973), 105–109
2 ERDOS, P , AND RENYI, A On random matrices *Magyar Tud Akad Mat Kutató Int. Kozl 8* (1964), 455–461 Reprinted in Erdos, P *The Art of Counting*, J Spencer, Ed , M I T Press, Cambridge, Mass (1973), pp 625–631
3 FROBENIUS, G Uber zerlegbare Determinaten *Sitzungsberichte der Berliner Akademie* (1917), 274–277
4 GONNET, G H Interpolation and interpolation hash searching Res Rep 76-02, Comptr Sci Dept , U of Waterloo, Waterloo, Ont , 1976
5 GONNET, G , AND MUNRO, I The analysis of an improved hashing technique Proc Ninth Annual ACM Symp on Theory of Comptng , Boulder, Colo , 1977, pp 113–121
6 KNUTH, D E *The Art of Computer Programming, Vol 3 Sorting and Searching* Addison-Wesley, Reading, Mass , 1973
7 KONIG, D Graphok és matrixok *Matematikai és Fizikai Lapok 38* (1931), 116–119.
8 KUHN, H W The Hungarian method for the assignment problem *Naval Res Log Quart 2* (1955), 83–97
9 PETERSON, W W Addressing for random-access storage *IBM J Res and Develop 1* (1957), 130–146
10 RYSER, H J Combinatorial Mathematics Carus Math Mono #14, Math Assoc Amer , 1963
11 TRIPP, R *International Thesaurus of Quotations* Thomas Y Crowell, New York, 1970

RECEIVED JUNE 1976, REVISED JUNE 1977

Code Optimization Techniques for Embedded DSP Microprocessors

Stan Liao Srinivas Devadas
MIT Department of EECS
Cambridge, MA 02139

Kurt Keutzer Steve Tjiang Albert Wang
Synopsys, Inc.
Mountain View, CA 94043

Abstract—We address the problem of code optimization for embedded DSP microprocessors. Such processors (e.g., those in the TMS320 series) have highly irregular datapaths, and conventional code generation methods typically result in inefficient code. In this paper we formulate and solve some optimization problems that arise in code generation for processors with irregular datapaths. In addition to instruction scheduling and register allocation, we also formulate the accumulator spilling and mode selection problems that arise in DSP microprocessors. We present optimal and heuristic algorithms that determine an instruction schedule simultaneously optimizing accumulator spilling and mode selection. Experimental results are presented.

Keywords—code generation, optimization, digital signal processors

I. Introduction

An increasingly common micro-architecture for embedded systems is to integrate a microprocessor or microcontroller, a ROM and an ASIC all on a single IC. Such a micro-architecture can currently be found in many diverse embedded systems, e.g., FAX modems, laser printers, and cellular telephones.

The programmable component in embedded systems can be an application-specific instruction processor (ASIP), a general-purpose microprocessor such as the SPARC, a microcontroller such as the Intel 8051, or a digital signal processing (DSP) microprocessor such as the TMS320C25. This paper focuses on the DSP application domain, where embedded systems are increasingly used. Many of these systems use processors from the TMS320C2x, 56K or ADSP families, all fixed-point DSP microprocessors with irregular datapaths.

As the complexity of embedded systems grow, the need to decrease development costs and time to market mandates the use of high-level languages (HLLs) in programming DSP processors; only short, time-critical portions of the program can be assembly-coded. Recent statistics from Dataquest support this trend: high-level languages (HLLs) such as C (and C++) are gradually replacing assembly language, because using HLLs greatly lowers the cost of development and maintenance of embedded systems. However, current compilers for fixed-point DSP microprocessors generate poor code — thus programming in a HLL can incur significant code performance and code size penalties.

While optimizing compilers have proved effective for RISC processors, the irregular datapaths and small number of registers found in DSP processors remain a challenge to compilers. The direct application of conventional code optimization methods (e.g., [1]) has, so far, been unable to generate code that efficiently uses the features of fixed-point DSP microprocessors.

Code size matters a great deal in embedded systems since program code resides in on-chip ROM, the size of which directly translates into silicon area and cost. Designers often devote a significant of time to reduce code size so that the code will fit into available ROM; exceeding on-chip ROM size could require expensive redesign of the entire IC [6]. As a result, a compiler that automatically generates small, dense code will result in a significant productivity gain as well.

There has been relatively little previous work in the area of code generation for DSP processors. Cheng and Lin present methods for code generation for the TMS320C40 in [2]. The algorithms they use are similar to high-level scheduling and allocation methods. Various groups in the hardware design community have recently started working on the problem of retargetable code generation for embedded processors [9]; the focus, however, has been on horizontally microcoded architectures.

In this paper, we develop code optimization techniques for DSP microprocessors with irregular datapaths that improve code performance and reduce code size. Our techniques are applicable to a broad class of DSP microprocessors, including those in the TMS320, DSP56K, and ADSP series. The optimization problems we target include instruction scheduling and register allocation. We also formulate the *accumulator spilling* and *mode selection* problems that arise in DSP microprocessors. The individual problems of scheduling and register allocation have traditionally been solved independently in compilers. We present optimal and heuristic algorithms that determine an instruction schedule while simultaneously optimizing accumulator spilling and mode selection. The experimental results we have obtained show significant improvements over existing code generation methods.

The paper is organized as follows. We use the TMS320C25 to illustrate our architecture model in Section II. In Section III we formulate the mode selection problem. We describe the general register allocation problem in Section IV and go on to formulate the accumulator spilling problem that is specific to DSP microprocessors. A branch-and-bound scheduling algorithm that produces a schedule for a basic block with a minimal number of accumulator spills and mode switches is presented in Section V. Experimental results are presented in Section VI. We conclude in Section VII with our ongoing research.

II. Example: TMS320C25

We describe Texas Instruments' popular TMS320C25 DSP microprocessor [7], highlighting the key features of this irregular architecture that are not addressed in traditional compiler optimizations. Fig. 1 shows a simplified model of its datapath.

The TMS320C25 is an accumulator-based machine. In addition to the usual ALU, there is a separate multiplier which takes input from the T register and memory and places the result in the P register. With the separate multiplier the machine can execute one-cycle multiply-accumulate operations. Note that there are no general-purpose registers other than the accumulator. Most operations involve an operand taken from the memory.

32nd ACM/IEEE Design Automation Conference ®

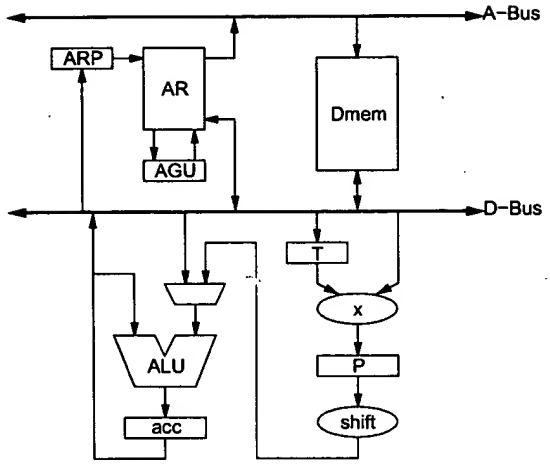


Fig. 1. TMS320C25 datapath (simplified model)

The memory is addressed by the address register file (AR0 through AR7), which is in turn addressed by the 3-bit address register pointer (ARP) (denoting the "current" AR). The address generation unit (AGU) allows the current AR to be auto-incremented or auto-decremented during the execution of any TMS320C25 instruction that uses indirect addressing mode.

Another feature of the TMS320C25 that is usually missing in general register machines is the use of *modes* (or *residual control* in microprogramming terminology). The most commonly used mode classes are sign-extension and product-shift. Some instructions are affected by the setting of the modes, and if the current mode setting is different from that desired by the instruction, then it must be set to the appropriate values.

The use of address registers and modes is likely intended to allow for more compact code. However, this means that the compiler's job is made more difficult. In the subsequent sections we will examine the problem of mode settings and present a formulation for the generalized problem. We will then extend this framework for the problem of minimizing the spills of the accumulator.

III. Mode Optimization Problem

The goal of mode optimization is to schedule the instructions so that the number of mode-setting instructions is minimized.

A. Simple Mode Optimization

Let the DAG $G = \langle V, E \rangle$ for a basic block be given. Let r be the number of modes, and $l : V \rightarrow \{1, \ldots, r\}$ label each node $v \in V$ with a mode $l(v)$. Let $C = [c_{ij}]$ be the cost matrix, where $c_{ij} \geq 0$ is the cost of switching from mode i to mode j. We assume that the following hold for the cost matrix C:

$$c_{ii} = 0, \quad \text{for all } i \tag{1}$$

$$c_{ik} \leq c_{ij} + c_{jk}, \quad \text{for all } i, j, k \tag{2}$$

Inequality (2) (triangular inequality) will be used later to establish lower bounds in the branch-and-bound algorithm. The *simple mode optimization problem* (SMOPT) is the problem of finding a linear schedule S that is a topological sort of G, $(v_1, v_2, \ldots, v_n)$, such that

$$\text{mode\_cost}(S) = \sum_{i=1}^{n} c_{l(v_i)l(v_{i+1})} \tag{3}$$

is minimized.

Theorem 1: The decision problem for simple mode optimization is NP-complete.

B. Multiple Mode Classes and Don't-Cares

In Section III-A we only considered a single mode class. A *mode class* is a set of mutually exclusive modes, and at any point the machine can only be in exactly one of the modes. For example, sign extension and product shift are two mode classes in the TMS320C25. We assume that each mode class has a cost matrix whose values are independent of every other mode class. In the absence of *don't-cares*, multiple mode classes are equivalent to a single mode class: their Cartesian product.

Let us first consider the presence of don't-cares (denoted by $-$) in a single mode class. A node is labeled $-$, if it is not affected by the current mode. Hence, we can first disregard these nodes, try to optimally schedule the other nodes, and then put these nodes back in the schedule, consistently with the original DAG. To do so, we construct from G a *reduced* DAG $G' = \langle V', E' \rangle$ as follows:

Procedure: Don't-Care Reduction

1. For each path $v_1, v_2, \ldots, v_{k-1}, v_k$ in G where $l(v_j) = -$, for $2 \leq j \leq k-1$, and $l(v_1), l(v_k) \neq -$, we add an edge (v_1, v_k). This preserves the precedence relationship from G.
2. Remove each node $v \in V$ such that $l(v) = -$, and all edges incident on or emanating from v.

Now let us consider m mode classes. The label of each node is an m-tuple, some or all of whose components may be $-$. For each mode class p we can construct a reduced DAG from G by projecting the label to the corresponding component and then reducing the graph using the above procedure; this derived graph is called the p-reduced DAG. We may then schedule these reduced DAGs separately. Each of these schedules is called a *reduced schedule.*

Definition 1: Let S_p and S_q be valid reduced schedules for the p- and q-reduced DAGs. S_p and S_q are said to be *compatible* if they can be merged to form a schedule that is valid with respect to G.

For example, the reduced schedules *ABEG* and *AEFG* are compatible; whereas *ABEG* and *AGDE* are not, since they have conflicting orders for nodes E and G. We state without proof the following proposition.

Proposition 1: Let S_p be an optimal reduced schedule for the p-reduced DAG, $p = 1, \ldots, m$. If all S_p's are compatible, then they can be merged to form an optimal schedule for G.

Unfortunately, not all optimal reduced schedules are compatible. Thus, if reduced DAGs are employed in a heuristic algorithm, some non-optimal reduced schedules will have to be chosen in order to achieve compatibility. In a branch-and-bound algorithm, we consider all modes classes simultaneously and compute the lower-bound by resolving the don't-cares conservatively.

C. Example

Let us consider the expression DAG G shown in Fig. 2(a). There are two mode classes, p={u,s} and q={0,1,2}. The corresponding reduced DAGs are shown in Fig. 2(b) and (c).

Assuming that any mode change incurs a unit cost, there are two optimal reduced schedules for the p-reduced DAG: S_{p1} =*ACBEF* and S_{p2} =*BACFE*. There is only one optimal reduced schedule for the q-reduced DAG: S_q =*ADFHEG*. S_{p1} and S_q are not compatible, since the former schedules F after E whereas the latter schedules F before E. However, S_{p2} and S_q are compatible, we obtain the optimal schedule for G: *BACDFHEG*.

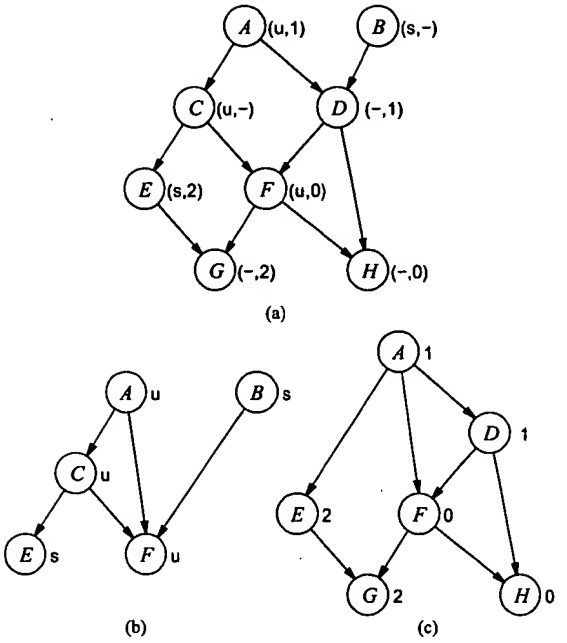


Fig. 2. (a) An expression DAG G. (b) p-reduced DAG. (c) q-reduced DAG.

IV. Register Allocation and Accumulator Spilling

This section describes the effect of the scheduling step on the number of values that have to be stored to perform a given computation. We first focus on conventional architectures and then switch focus to irregular datapaths such as the TMS320C25.

A. RISC Architectures

Register allocation deals with allocating variables in the given basic block to a minimum number of registers. If at any time we have to store a set of values whose cardinality exceeds the number of available registers, *spilling* into memory is the only alternative. The register allocation problem for RISC machines is quite different from machines such as the TMS320C25 since the latter does not have a register file. However, in both cases it is possible to arrive at a cost function for scheduling that simplifies the succeeding register allocation step.

B. Effect of Scheduling on Register Allocation

It is well known that different schedules corresponding to a data-dependency graph require differing number of registers. We formalize this effect in the sequel.

We are given a basic block represented as a DAG $G = \langle V, E \rangle$. We will assume a sequential model of processor execution, however, the discussion can easily be generalized to include parallelism corresponding to multiple execution threads. Each node $v \in V$ is assumed to have two input variables $i_1(v)$ and $i_2(v)$ and an output variable $o(v)$. We further assume the graph is in static single assignment form, that is, every assignment is to a unique variable.

If we schedule the nodes in G, the the life-times of all the variables can be calculated. The **life-time** of a variable is the duration between its unique assignment and last use. Since most processors allow the reading and writing of values into a register in a single instruction, if a variable is written in instruction i and is read in instruction $j > i$, we will denote its life-time as the (open-ended) interval $[i, j)$.

```
v1 = v2 + v3
v4 = v2 - v3
v5 = v1 * v2
v6 = v4 & v3
v7 = v5 | v6
```

(a)

```
R1 = R2 + R3
R4 = R2 - R3
R1 = R1 * R2
R4 = R4 & R3
R4 = R1 | R4
```

(b)

```
R1 = R2 + R3
R1 = R1 * R2
R2 = R2 - R3
R2 = R2 & R3
R3 = R1 | R2
```

(c)

Fig. 3. (a) Code sequence (b) Register allocation on original code sequence (c) Register allocation on reordered code sequence

Variables with non-overlapping life-times can be merged into the same register. For example, the variable $i_1(v)$ with life-time $[i, j)$ can be merged with variable $i_2(v)$ with life-time $[k, l)$ if $k \geq j$.

The number of registers required is proportional to the overlap of the live periods of the variables, or to put it differently, the number of registers required is the *maximal density* of variable life-times across the entire sequence. Given a set of variable life-times in a schedule we can compute the maximal density in linear time.

The *simple register allocation problem* is to find the best possible grouping of variables with non-overlapping life-times into a minimum number of sets. Given a fixed schedule the simple register allocation problem (SRAOPT) can be solved in polynomial time. (A polynomial-time solution is only possible when static single assignment for each variable is assumed, in which case the interference graph is an *interval graph*, which can be colored in polynomial time [3].) However, there is freedom in the ordering of the nodes of the given DAG as long the dependency constraints are not violated. Given a code sequence, exploiting this freedom can result in a smaller set of registers being required. This is illustrated in Fig. 3. In Fig. 3(a), an example code sequence being executed on a processor with a single arithmetic unit is shown. Without changing the order of the operations in the code sequence, the minimum number of registers required is 4, as shown in Fig. 3(b). Allowing re-ordering of operations within the sequence produces a 3 register solution in Fig. 3(c). (A similar example was given in [10].)

Finding the optimal ordering of operations within a sequence, so as to allocate a minimum set of registers reduces to a one-dimensional linear arrangement problem similar to SMOPT of Section III. The *register allocation problem* (RAOPT), involves finding a schedule S that is a topological sort of a basic block represented by a DAG G, $(v_1, v_2, \ldots, v_n)$, such that

$$\text{reg\_cost}(S) = \max_{i=1}^{n} \text{density}(i) \tag{4}$$

is minimized, where density(i) corresponds to the number of variables that are live at instruction i.

C. DSP Microprocessor Architectures

Register allocation for processors with a general-purpose register file is relatively straightforward. Obtaining a schedule that minimizes the maximal density of variable life-times will result in minimal spilling into memory.

For processors such as the TMS320C25, the register allocation problem is more complicated due to several reasons, the foremost of which is the indirect addressing mechanism that is used in the datapath.

- The TMS320C25 has a single accumulator and no general-purpose registers.
- The address registers AR0 through AR7 are in turn addressed by the register ARP.
- The address registers AR0 through AR7 can be auto-incremented and auto-decremented during the execution of any TMS320C25 instruction that uses indirect addressing mode. However, loading a new address requires a separate instruction.
- The ARP can be switched to point to a different address register during any instruction using indirect addressing mode.

All of the above complications can be taken into account by formulating the register allocation problem for a fixed code schedule as an *offset assignment problem* [8]. In this paper, we will focus on the first item above, and formulate the minimal spilling problem for accumulator-based architectures.

In the TMS320C25, instructions such as ADD, MAC and LAC (load-accumulator) write into the accumulator. The MPY instruction writes into the P register. Given a schedule it is easy to determine the number of accumulator spills using life-time analysis. If the accumulator value is used in the immediately following instruction and nowhere else, then spilling into memory is not required, else we need to spill the contents into memory for later access. Different schedules will result in different numbers of accumulator spills.

The *accumulator spilling problem* (ASOPT), involves finding a linear schedule S that is a topological sort of a basic block represented by a DAG G, $(v_1, v_2, \ldots, v_n)$, such that

$$\text{spill\_cost}(S) = \text{Number of accumulator spills in } S \qquad (5)$$

is minimized.

V. A Branch-and-Bound Algorithm for Scheduling

In this section we present a branch-and-bound algorithm which, given a basic block represented as a DAG, determines an optimal code schedule under a specified cost function.

A. Cost Function

The cost we use includes the number of accumulator spills required by the schedule S (see Eqn. (5)) as well as the number of mode switches required (see Eqn. (3)).

The cost function we use in the branch-and-bound method is:

$$C(S) = W_S \times \text{spill\_cost}(S) + W_M \times \text{mode\_cost}(S)$$

where W_S and W_M depend on the relative cost of instructions required to spill the accumulator to memory and instructions required to accomplish a mode switch.

B. Branching Search

Branching over all possible solutions is accomplished using the recursive branching strategy of Fig. 4. Initially, **find-optimal-schedule()** is called with the original DAG $G = \langle V, E \rangle$, and $P = \phi$ as parameters. It returns the best schedule S and the cost of the schedule $C(S)$.

```
find-optimal-schedule( G, P ):
{
    /* G = DAG of basic block */
    /* P = Current partial schedule for DAG */
    /* C(P) = Cost of partial schedule */
    N = find-scheduleable-nodes( G, P );
    if ( N ≡ φ ) {
        if ( C(P) < C(S) )
            [ S, C(S) ] = [ P, C(P) ];
            return [ S, C(S) ];
    }

    foreach node v in N {
        LB = lower-bound( G − v, P ∪ v );
        if ( C(P ∪ v) + LB < C(S) ) {
            [ T, C(T) ] = find-optimal-schedule( G − v, P ∪ v );
            if ( C(T) < C(S) )
                [ S, C(S) ] = [ T, C(T) ];
        }
    }
    return [ S, C(S) ];
}
```

Fig. 4. Branch-and-bound procedure to determine optimal schedule

The procedure **find-scheduleable-nodes()** determines the set of nodes $N \in V$ that can be scheduled given the partial schedule P such that dependency constraints are not violated. If there are no scheduleable nodes it means that the schedule is complete. If the cost of this complete schedule is less than the best cost seen thus far, we save the complete schedule P as the best schedule and return to the previous level of recursion.

If there are scheduleable nodes in N, we select each of them in sequence and recursively call **find-optimal-schedule()**. Once we have chosen a node v to add to P, we first compute a lower bound on the cost of any schedule we will see in this recursion path. If the cost of the partial schedule $P \cup v$ plus the computed lower bound on the unscheduled DAG $G - v$ is greater than or equal to the best cost seen thus far, there is no need to explore this recursion path. The best schedule with its associated cost is returned by procedure **find-optimal-schedule()**.

C. Lower Bound Computation

The procedure **lower-bound()** is critical to improving the efficiency of the search. If we can compute tight lower bounds, then we can prune the search considerably by reducing the depth of recursion.

Given a DAG $\tilde{G}\langle V, E\rangle$ we need to compute a lower bound over all possible schedules $\tilde{P}$ consistent with $\tilde{G}$. This entails computing a lower bound for mode_cost($\tilde{P}$) and a lower bound for spill_cost($\tilde{P}$).

C.1 Lower Bound for Spill Cost

We assume that the accumulator has no useful value upon entry to the basic block in Eqn. (5) as well as in the lower bound computation below.

We mark the nodes in the given DAG $\tilde{G}$ using the steps below. Initially all nodes are unmarked.

1. Nodes whose outputs are outputs of the basic block and which write into the accumulator (e.g., MAC and ADD) will spill their contents, and these nodes are marked.

2. If a node v in the $\tilde{G}$ has more than 2 fanouts, it means that $o(v)$ is used as an input in two other instructions and this implies that the accumulator contents corresponding to $o(v)$ has to be spilled into memory.
3. If node v receives inputs from nodes x and y which correspond to instructions that write into the accumulator, then either $o(x) = i_1(v)$ or $o(y) = i_2(v)$ has to be spilled. Therefore, if both x and y are unmarked, we will mark x or we will mark y (but not both).

The number of marked nodes in $\tilde{G}$ corresponds to a lower bound on the number of accumulator spills in any schedule consistent with $\tilde{G}$.

C.2 Lower Bound for Mode Cost

To estimate a lower bound for the mode cost given an unscheduled DAG $\tilde{G}$, we simply compute the maximum cost for mode switching using Eqn. (3) along *any path* of $\tilde{G}$. This follows from Inequality (2), since any schedule must contain this path as a subsequence, and the cost of switching from mode $i \rightarrow j$ cannot be greater than that of mode $i \rightarrow j \rightarrow k$ for any k (for otherwise we can replace the former by the latter).

D. Hashing

The branching procedure of Fig. 4 may perform a significant amount of redundant computation. Consider a situation where we have constructed a partial schedule P_1 which corresponds to the set of nodes V_1. The optimal scheduling subproblem is then solved for $G - V_1$ with appropriate initial conditions corresponding to the mode and accumulator contents of the last instruction in P_1. Now, if we compute a different partial schedule P_2 corresponding to the same set of nodes V_1, and the last instruction in P_2 is the same as the last instruction in P_1, we solve exactly the same optimal scheduling subproblem for $G - V_1$. However, there is no mechanism in the procedure of Fig. 4 to detect that we have solved the subproblem for $G - V_1$ already.

The lower bounding technique alleviates the above inefficiency to a certain extent. However, since the bounds may not always be tight, significant redundant computation may occur.

A hashing mechanism of "remembering" previously computed optimal solutions for parts of the original DAG can greatly improve the efficiency of the search. We hash each partial schedule P_i such that $|P_i| \leq L$, where L is a user-specified parameter in the range $2 \leq L \leq |V|$. The hash is computed in such a way that if different partial schedules P_i and P_j contain the same set of nodes V_1, the same hash is computed. Once the subproblem of finding an optimal schedule for $G - V_1$ has been solved for P_i, we store the result in a hash table that is accessible by the hash for P_i. Before we begin to solve the subproblem associated with P_j, we check to see if $|P_j| \leq L$, if so we compute the hash for P_j and access the hash table. If we get a "hit" in the hash table, we immediately return the best solution for the subproblem of scheduling $G - V_1$. A hit in the hash table implies that P_i and P_j correspond to the same set of instructions, and further that the last instructions in P_i and P_j are the same.

E. Heuristics

Once the nodes in N have been determined by the procedure **find-scheduleable-nodes**(), we can recursively call **find-optimal-schedule**() for each of the nodes in N in any order. However, to improve efficiency it is worthwhile to first explore partial solutions that have a good chance of being extended to optimal solutions. This determination can only be made heuristically.

We sort the nodes in N based on a cost estimate to obtain a sorted list $sort(N)$. The cost estimate for any $v \in N$ is equal to $C(P \cup v) -$ $C(P)$ which includes both the spill cost and the mode switching cost. Nodes in N are sorted in increasing order of this cost estimate.

For small to moderate-sized basic blocks the optimal algorithm of Fig. 4 that explores all possible solutions is viable. For large basic blocks, we have to resort to heuristic search techniques. A fast, greedy heuristic is based on the node sorting method described above. We only explore solutions corresponding to the first t nodes in the sorted list $sort(N)$. The **foreach** loop of Fig. 4 is replaced with t calls to **find-optimal-schedule**(), where typically $1 \leq t \leq 3$.

basic block	fixed costs			orig. sched.		
	I	C	V	L	S	M
SpeedCtl386	11	0	5	6	1	10
SpeedCtl389	11	0	6	7	1	8
Compaction3	13	0	6	12	6	6
FFTBR28	22	4	9	17	4	11
ChenDct1	78	20	17	47	11	25
ChenDct4	80	20	17	46	11	25
ChenIDct1	78	26	10	55	20	33
ChenIDct4	72	18	10	39	12	17
LeeDct1	76	17	25	45	4	26
LeeDct4	50	8	21	31	2	16
LeeIDct1	79	26	10	49	17	32
LeeIDct4	64	15	20	35	2	21

TABLE I
FIXED COSTS AND ORIGINAL SCHEDULE

basic block	heur. sched.			ratio (LSM)	ratio (all)
	L	S	M		
Speedctl386	5	0	8	0.764	0.879
Speedctl389	6	0	6	0.750	0.879
Compaction3	7	1	2	0.417	0.674
FFTBR28	12	2	6	0.625	0.820
ChenDct1	32	6	6	0.530	0.803
ChenDct4	31	6	6	0.524	0.804
ChenIDct1	29	6	5	0.370	0.694
ChenIDct4	29	6	5	0.588	0.833
LeeDct1	38	5	2	0.600	0.844
LeeDct4	26	1	2	0.592	0.844
LeeIDct1	25	2	2	0.296	0.676
LeeIDct4	27	2	4	0.569	0.840

TABLE II
HEURISTIC SCHEDULE AND RATIOS

VI. EXPERIMENTS AND RESULTS

We have implemented the heuristic algorithm of Section V-E to perform scheduling for minimum cost. Our experiments are based on a code generator for a simplified TMS320C25 architecture with all the features described in Section II. A full-featured TMS320C25 code generator is currently under development (see Section VII).

To accurately account for the various types of costs, we attribute the following cost components to each node.

- Instruction (I). Each node in the DAG is associated with an instruction that has a cost of 1.
- Common subexpression (C). If the node has two or more uses, it is

a common subexpression. Under our assumption of aggressive common subexpression elimination, the result of this node is stored to memory rather than be recomputed at a later time.
- Live-on-exit variable (V). The result of the computation for this node is live upon exit of this basic block.
- Load (L). One operand of the node is not in the accumulator; therefore, it needs to be loaded.
- Spill (S). The result of the previously scheduled node will be used later but not now.
- Mode change (M). The node requires a mode setting different from the current setting. The mode classes considered are sign-extension and product-shift.

The first three items are fixed costs. Under any schedule, the number of instructions, common subexpressions, and live-on-exit variables remains the same. Therefore, the only optimizable costs are those of loads, spills, and mode changes.

We present our experimental results in Tables I and II. The former gives the original schedule generated by the front-end after common subexpression elimination. This schedule closely resembles that found in the source code. The latter shows the results we have obtained after our heuristic scheduling.

SpeedCtl is a routine in an ADPCM transcoder applying the CCITT recommendation G.721. Compaction is a notch-filter routine. FFTBR is an Fast Fourier Transform routine with a mechanism to prevent overflows. These three are taken from the DSPstone benchmark suite [11]. ChenDct, ChenIDct, LeeDct, and LeeIDct are discrete cosine transform routines in a JPEG package. We have chosen the largest basic blocks from these routines. The column labeled "ratio (LSM)" gives the ratio of only the optimizable costs (loads, spills, and mode changes), where as the "ratio (all)" column gives the ratio with the fixed costs (I, C, and V) taken into account. These results are very encouraging. Even with the simple heuristic we were able to achieve substantial improvement over the schedule given by the front-end.

VII. Conclusions and Ongoing Work

Code generation for irregular datapaths, such as those used in DSP microprocessors, is a problem that has received relatively little attention to date. With the advent and increasing use of embedded systems, this problem has become very important. In this paper we presented scheduling algorithms that are able to exploit the features of the TMS320C25 microprocessor. Our initial results indicate that these algorithms obtain substantial improvements in code size and performance over conventional code generation techniques.

We are currently developing a framework for retargetable code generation [9]. There are many avenues for further work in this area. Our framework is directly applicable to traces [4][5] rather than just basic blocks, and experiments on traces will be conducted in the near future. Traces will allow for more global optimization and afford the possibility of even greater savings over conventional optimization. One way to avoid the possible code explosion caused by trace scheduling is to restrict the movement across basic blocks to mode-setting instructions. This way we ensure that along the most frequent traces the number of such instructions is minimized.

The framework can be easily generalized to accumulator-based machines which also have a general-purpose register file such as the TMS320C40. This can easily be done by adding Eqn. (4) to the cost function.

Storage assignment [8] is a very important post-scheduling problem that has to be solved in order to ensure that memory accesses have minimal cost. For machines (such as the TMS320C25) without indexing addressing mode, variables are accessed through address registers (the ARs) and it is desirable that the auto-increment/decrement feature be efficiently utilized. The placement of variables in storage has a significant impact on the size and performance of the generated code. For instance, if variables are accessed in the order `abacd` and the following assignment is made: a:1, b:2, c:3, and d:4, accessing a followed by c requires an explicit instruction to increase the AR by two; every other access can be accomplished using auto-increment or decrement. On the other hand, if we use the assignment: a:2, b:1, c:3, and d:4, then all changes in the AR can be done via auto-increment or decrement; no explicit instruction for changing the AR is necessary. This problem is related to the mode optimization problem in that the AR can be considered a mode class and our goal is again to minimize the number of "mode-setting" instructions. Its relationship with scheduling is, however, much more complicated because before the actual assignment is made, the information is only symbolic, and it is very difficult to estimate the effect of scheduling on offset assignment.

Finally, to fully exploit the features of many DSP microprocessors, zero-overhead loops have to be detected and appropriate code generated wherever possible.

VIII. Acknowledgements

This research was supported in part by the Advanced Research Projects Agency under contract DABT63-94-C-0053, and in part by a NSF Young Investigator Award with matching funds from Mitsubishi Corporation.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] W-K. Cheng and Y-L. Lin. Code Generation for a DSP Processor. In *Proceedings of the Int'l Symposium on High-Level Synthesis*, pages 82–87, May 1994.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[4] John R. Ellis. *A Compiler for VLIW Architectures*. MIT Press, 1985.

[5] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trnsactions on Computers*, C-30(7):478–490, 1981.

[6] J. G. Ganssle. *The Art of Programming Embedded Systems*. San Diego, CA: Academic Press, Inc., 1992.

[7] Texas Instruments. *TMS320C2x User's Guide*. Texas Instruments, January 1993. Revision C.

[8] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage Assignment to Decrease Code Size. In *Proceedings of ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[9] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995. Proceedings of the 1994 Dagstuhl Workshop on Code Generation for Embedded Processors.

[10] S. S. Pinter. Register Allocation with Instruction Scheduling: a New Approach. In *Proceedings of the ACM Programming Language Design and Implementation Conference*, pages 248–257, June 1993.

[11] V. Živojnović, J. Martínez Velarde, and C. Schläger. DSPstone: A DSP-oriented Benchmarking Methodology. Technical report, Aachen University of Technology, August 1994.

Fundamental Techniques for Order Optimization

David Simmen
IBM Santa Teresa Lab
dsimmen@vnet.ibm.com

Eugene Shekita
IBM Almaden Research Center
shekita@almaden.ibm.com

Timothy Malkemus
IBM Austin Lab
malkemus@vnet.ibm.com

Abstract

Decision support applications are growing in popularity as more business data is kept on-line. Such applications typically include complex SQL queries that can test a query optimizer's ability to produce an efficient access plan. Many access plan strategies exploit the physical ordering of data provided by indexes or sorting. Sorting is an expensive operation, however. Therefore, it is imperative that sorting is optimized in some way or avoided all together. Toward that goal, this paper describes novel optimization techniques for pushing down sorts in joins, minimizing the number of sorting columns, and detecting when sorting can be avoided because of predicates, keys, or indexes. A set of fundamental operations is described that provide the foundation for implementing such techniques. The operations exploit data properties that arise from predicate application, uniqueness, and functional dependencies. These operations and techniques have been implemented in IBM's DB2/CS.

1 Introduction

As the cost of disk storage drops, more business data is being kept on-line. This has given rise to the notion of a data warehouse, where non-operational data is typically kept for analysis by decision support applications. Such applications typically include complex SQL queries that can test the capabilities of an optimizer. Often, huge amounts of data are processed, so an optimizer's decisions can mean the difference between an execution plan that finishes in a few minutes verses one that takes hours to run.

Many access plan strategies exploit the physical ordering of data provided by indexes or sorting. Sorting is an expensive operation, however. Therefore, it is imperative that sorting is optimized in some way or avoided all together. This leads to a non-trivial optimization problem, however, because a single complex query can give rise to multiple *interesting orders* [SAC+79]. Here, an interesting order refers to a specification for any ordering of the data that may prove useful for processing a join, an ORDER BY, GROUP BY, or DISTINCT. To be effective, an optimizer must detect when indexes provide an interesting order, the optimal place to sort if sorting is unavoidable, the minimal number of sorting columns, whether two or more interesting orders can be combined and satisfied by a single sort, and so on. This process will be referred to as *order optimization*.

At first glance, it might seem like hash-based set operations [BD83, DKO+84] make order optimization a non-issue, since hash-based operations do not require their input to be ordered. An index may already provide an interesting order for some operation, however, making the hash-based alternative more expensive. This is particularly true in warehousing environments, where indexes are pervasive. As a result, an optimizer needs to be cognizant of interesting orders. It should always consider both hash- and order-based operations and pick the least costly alternative [Gra93].

Although people have been building SQL query optimizers for close to twenty years [JV84, Gra93], there has been surprisingly little written about the problem of order optimization. This paper describes novel techniques to address that problem. One of the paper's key contributions is an algorithm for reducing an interesting order to a simple canonical form by using applied predicates and functional dependencies. This is essential for determining when sorting is actually required. Another important contribution is the notion of *sort-ahead*, which allows a sort for something like an ORDER BY to be pushed down in a join tree or view. All of these techniques have been implemented in the query optimizer of IBM's DB2/CS, which is the client-server version of DB2 that runs OS/2, Microsoft Windows NT, and various flavors of UNIX. Henceforth, DB2/CS will be referred to as simply DB2. Much of the discussion in this paper is framed in the context of the DB2 query optimizer. The techniques that are described have general applicability, however, and could be used in any query optimizer.

The remainder of this paper is organized as follows: In Section 2, related work is described. This is followed by a brief overview of the DB2 optimizer in Section 3. Next, fundamental operations for order optimization are

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

described in Section 4. In Section 5, the architecture of the DB2 optimizer that has been built around those fundamental operations is described. An example is then provided in Section 6 to illustrate how things tie together. Advanced issues beyond the scope of this paper are mentioned in Section 7. Finally, performance results are presented in Section 8, and conclusions are drawn in Section 9.

2 Related Work

The classic work on the System R optimizer by Selinger et al. [SAC+79] was the first research to look at the problem of order optimization. That paper coined the term "interesting orders". In System R, interesting orders were mainly used to prevent subplans that satisfy some useful order from being pruned by less expensive but unordered subplans during bottom-up plan generation.

A recent paper on the Rdb optimizer [Ant93] talked about combining interesting orders from ORDER BY, GROUP BY, and DISTINCT clauses, if possible, so at most one sort could be used. That paper was primarily an overview of the Rdb optimizer, however. It did not specifically focus on order optimization.

Other, more loosely related papers include those on predicate migration [Hel94] and group-by push-down [YL93, CS93]. Predicate migration considers whether an expensive predicate should be applied before or after a join. Similarly, group-by push-down considers whether GROUP BY should be performed before a join. In each case, an optimizer determines which is the better alternative using its cost estimates. Both techniques are similar to the notion of sort-ahead, as described in this paper.

3 Overview

The DB2 optimizer is a direct descendent of the Starburst optimizer described in [Loh88, HFLP89]. Among other things, the DB2 optimizer uses much more sophisticated techniques for order optimization. This section provides an overview of the DB2 optimizer to establish some background and terminology. More details will be given later.

The DB2 optimizer actually has several distinct optimization phases. Here, we are mainly concerned with the phase where traditional cost-based optimization occurs. Prior to this phase, an input query is parsed and converted to an intermediate form called the *query graph model* (QGM).

The QGM is basically a high-level, graphical representation of the query. *Boxes* are used to represent relational operations, while arcs between boxes are used to represent *quantifiers*, i.e., table references. Each box includes the predicates that it applies, an input or output order specification (if any), a distinct flag, and so on. The basic set of boxes include those for SELECT, GROUP BY, and UNION. Joins are represented by a SELECT box with two or more input quantifiers, while ORDER BY is represented by a SELECT box with an output order specification.

After its construction, the original QGM is transformed into a semantically equivalent but more "efficient" QGM using heuristics such as predicate push-down, view merging, and subquery-to-join transformation. [PHH92]. Finally, cost-based optimization is performed. During this phase, the QGM is traversed and a *query execution plan* (QEP) is generated.

A QEP can be viewed as a dataflow graph of *operators*, where each node in the graph corresponds to a relational operation like a join or a low-level operation like a sort. Each operator consumes one or more input records (i.e., a table), and produces an output set of records (another table). We will refer to these as input and output *streams*. Figure 1 illustrates what the QGM and QEP might look like for a simple query.

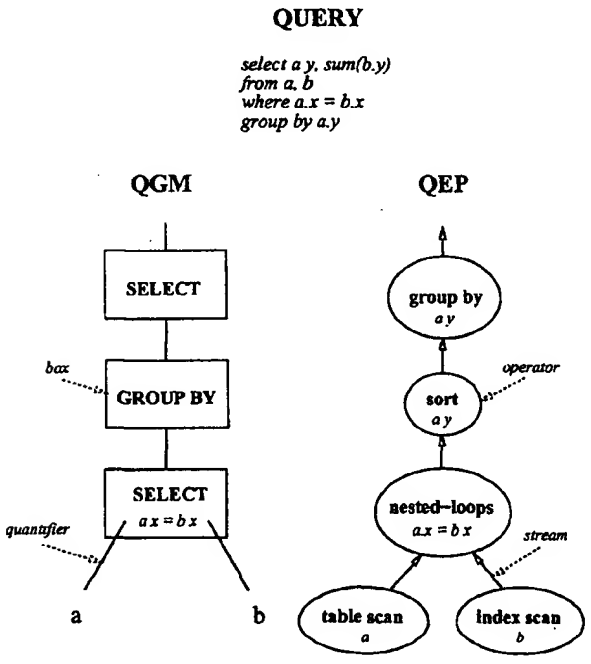


Figure 1: Simple QGM and QEP Example

Each stream in a QEP has an associated set of *properties* [GD87, Loh88]. Examples of properties include the columns that make up each record in the stream, the set of predicates that have been applied to the stream, and the order of the stream. Each operator in a QEP determines the properties of its output stream. The proper-

ties of an operator's output stream are a function of its input stream(s) and the operation being applied by the operator. For example, a sort operator passes on all the properties of its input stream unchanged except for the order property and cost. Note that a stream's order, if any, always originates from an ordered index scan or a sort.

During the *planning phase* of optimization, the DB2 optimizer builds a QEP bottom-up, operator-by-operator, computing properties as it goes. At each step, different alternatives are tried and more costly subplans with comparable properties are pruned [Loh88]. At strategic points during planning, the optimizer may decide to build a QEP which satisfies an interesting order. A sort may need to be added to a QEP if there is no existing QEP with an order property satisfying the interesting order.

Interesting orders are generated in a top-down scan of QGM prior to the planning phase. This is referred to as the *order scan* of QGM. Interesting orders arise from joins, ORDER BY, GROUP BY, or DISTINCT, and are hung off the QGM. Here, both order properties and interesting orders will be denoted as a simple list of columns in major to minor order, i.e., $(c_1, c_2, ..., c_n)$. Without loss of generality, we will always assume that an ascending order is required for each column c_i.

Interesting orders are pushed down and combined in the order scan whenever possible. This allows one sort to satisfy multiple interesting orders. As interesting orders are pushed down they can turn into *sort-ahead* orders. These allow the optimizer to try pushing down a sort for, say, an ORDER BY to an arbitrary level in a join tree. Different alternatives are tried, and only the least costly one is kept. The next section looks at the fundamental operations on interesting orders needed to accomplish these tasks.

4 Fundamental Operations for Order Optimization

4.1 Reduce Order

The most fundamental operation used by order optimization is something referred to as *reduction*. Reduction is the process of rewriting an order specification (i.e., an order property or interesting order) in a simple canonical form. This involves substituting each column in the specification with a designated representative of its equivalence class (called the *equivalence class head*) and then removing all redundant columns. Reduction is essential for testing whether an order property satisfies an interesting order.

As a motivating example, consider an arbitrary interesting order $I = (x, y)$, and suppose an input stream has the order property $OP = (y)$. A naive test would conclude that I is not satisfied by OP, and a sort would be added to the QEP. Suppose, however, that a predicate of the form $col = constant$ has been applied to the input stream, e.g., $x = 10$. Then the column x in I is redundant since it has the value 10 for all records. Hence, I can be rewritten as $I = (y)$. After being rewritten, it is easy to determine that OP satisfies I, so no sort is necessary. Note that a literal expression, host variable, or correlated column qualify as a constant in this context.

Reduction also needs to take column equivalence classes into account. These are generated by predicates of the form $col = col$. For example, suppose $I = (x, z)$ and $OP = (y, z)$. Further suppose that the predicate $x = y$ has been applied. The equivalence class generated by $x = y$ allows OP to be rewritten as $OP = (x, z)$. After being rewritten, it is easy to determine that OP satisfies I.

Reduction also needs to take keys into account. For example, suppose $I = (x, y)$ and $OP = (x, z)$. If x is a key, then these can be rewritten as $I = (x)$ and $OP = (x)$. Here, y and z are redundant since x alone is sufficient to determine the order of any two records.

Keys are really just a special case of functional dependencies (FDs) [DD92]. So rather than keys, FDs are actually used by reduction, since they are more powerful. In the DB2 optimizer, a set of FDs are included in the properties of a stream. The way FDs are maintained as a property will be discussed in more detail later.

The notation used for FDs is as follows: A set of columns $A = \{a_1, a_2, ..., a_n\}$ functionally determines columns $B = \{b_1, b_2, ..., b_m\}$ if for any two records with the same values for columns in A, the values for columns in B are also the same. This is denoted as $A \rightarrow B$. The *head* of the FD is A, while the *tail* is B.

It is important to note that all of the above optimizations can be framed in terms of functional dependencies. This is because a predicate of the form $x = 10$ gives rise to $\{\} \rightarrow \{x\}$, i.e., the "empty-headed" FD [DD92]. Moreover, a predicate of the form $x = y$ gives rise to $\{x\} \rightarrow \{y\}$ and $\{y\} \rightarrow \{x\}$. If $x = y$ is a join predicate for an outer join, then $\{x\} \rightarrow \{y\}$ holds if x is a column from a non-null-supplying side. In addition, $\{x\} \rightarrow \{all\ cols\}$ when x is a key. Finally, $\{x\} \rightarrow \{x\}$ is always true.

The mapping of predicate relationships and keys to functional dependencies makes it possible to express reduction in a very simple and elegant way. The algorithm for Reduce Order is shown in Figure 2. In the algorithm, note that the equivalence class head is chosen from those columns made equivalent by predicates already applied to the stream. Also note that $B \rightarrow \{c_i\}$ if there exists some $B' \rightarrow C$ where $B' \subseteq B$ and $C \supseteq \{c_i\}$. This follows from the algebra on FDs [DD92]. Consequently, simple subset operations can be used on the input FDs to test

whether $B \rightarrow \{c_i\}$.

Reduce Order
input:
  a set of FDs, applied predicates, and
  order specification $O = (c_1, c_2, ..., c_n)$
output:
  the reduced version of O

1) rewrite O in terms of each column's
   equivalence class head
2) scan O backward
3) for (each column c_i scanned)
4) let $B = \{c_1, c_2, ..., c_{i-1}\}$, i.e.,
    the columns of O preceding c_i
5) if ($B \rightarrow \{c_i\}$) then
6) remove c_i from O
7) endif
8) endfor

Figure 2: Reduce Order Algorithm

The correctness proof for Reduce Order is straightforward. Consider what happens when two records r_1 and r_2 are compared. The only time the value of c_i affects their order is when r_1 and r_2 have the same values for all columns in C. But then $r_1.c_i$ and $r_2.c_i$ must also have the same value because $B \rightarrow \{c_i\}$. Consequently, removing c_i will not change the order of records produced by O.

Before moving on, note that an order specification can become "empty" after being reduced. For example, suppose the predicate $x = 10$ has been applied and the interesting order $I = (x)$ is reduced. The predicate $x = 10$ gives rise to $\{\} \rightarrow \{x\}$. Consequently, I will reduce to the empty interesting order $I = ()$, which is trivially satisfied by any input stream.

4.2 Test Order

As it generates a QEP, the optimizer has to test whether a stream's order property OP satisfies an interesting order I. If not, a sort is added to the QEP. The algorithm for Test Order is shown in Figure 3. Note that when a sort is required, the reduced version of I provides the minimal number of sorting columns, which is important for minimizing sort costs.

4.3 Cover Order

As mentioned earlier, the DB2 optimizer tries to combine interesting orders in the top-down order scan of QGM. This often allows one sort to satisfy multiple interesting orders. When two interesting orders are combined, a *cover* is generated. The cover of two interesting

Test Order
input:
  an interesting order I and an order
  property OP
output:
  true if OP satisfies I, otherwise false

1) reduce I and OP
2) if (I is empty or the columns in I
     are a prefix of the columns in OP) then
3) return true
4) else
5) return false
6) endif

Figure 3: Test Order Algorithm

orders I_1 and I_2 is a new interesting order C such that any order property which satisfies C also satisfies both I_1 and I_2. For example, the cover of $I_1 = (x)$ and $I_2 = (x, y)$ is $C = (x, y)$.

Of course, it is not always possible to generate a cover. For example, there is no cover for $I_1 = (y, x)$ and $I_2 = (x, y, z)$. As in Test Order, however, interesting orders need to be reduced before attempting a cover. Suppose the predicate $x = 10$ has been applied in this example. Then the interesting orders would reduce to $I_1 = (y)$ and $I_2 = (y, z)$, giving the cover $C = (y, z)$. The algorithm for Cover Order is shown in Figure 4.

Cover Order
input:
  interesting orders I_1 and I_2
output:
  the cover of I_1 and I_2; or a return code
  indicating that a cover is not possible

1) reduce I_1 and I_2
2) *w.l.o.g.*, assume I_1 is the shorter interesting order
3) if (I_1 is a prefix of I_2) then
4) return I_2
5) else
6) return "cannot cover I_1 and I_2"
7) endif

Figure 4: Cover Order Algorithm

4.4 Homogenize Order

As mentioned earlier, an attempt is made to push down interesting orders in the order scan of QGM so that sort-ahead may be attempted. When an interesting order I is

pushed down, some columns may have to be substituted with equivalent columns in the new context. This is referred to as *homogenization*. For example, consider the following query:

```
select *
from a, b
where a.x = b.x
order by a.x, b.y
```

Here, the ORDER BY gives rise to the interesting order $I = (a.x, b.y)$. The order scan will try to push down I to the access of both table a and table b as a sort-ahead order. For the access of table b, the equivalence class generated by $a.x = b.x$ is used to homogenize I as $I_b = (b.x, b.y)$.

I cannot be pushed down to the access of table a, since $b.y$ is unavailable until after the join. However, suppose $a.x$ is a base-table key that remains a key after the join [DD92]. If so, $\{a.x\} \rightarrow \{b.y\}$. This allows I to be reduced to $I = (a.x)$, which can be pushed down to the access of table a. As this example illustrates, an interesting order needs to be reduced before being homogenized. The algorithm for Homogenize Order is shown in Figure 5.

Homogenize Order
input:
an interesting order I and target columns $C = \{c_1, c_2, ..., c_n\}$
output:
I homogenized to C, that is, I_C; or a return code indicating that I_C is not possible

```
1) reduce I
2) using equivalence classes, try to substitute each
   column in I with a column in C
3) if ( all the columns in I could be substituted ) then
4)    return I_C
5) else
6)    return "cannot homogenize I to C"
7) endif
```

Figure 5: Homogenize Order Algorithm

Note that unlike Reduce Order, Homogenize Order can choose any column in the equivalence class for substitution. Moreover, there is no need to choose from just the columns that have been made equivalent by predicates applied so far. Columns that will become equivalent later because of predicates that have yet to be applied can also be considered. This is because homogenization is concerned with producing an order that will eventually satisfy I.

5 The Architecture for Order Optimization in DB2

This section describes the overall architecture of the DB2 optimizer for order optimization. Only a high-level summary of the architecture is provided. The focus will be those parts of the architecture that have been built around the fundamental operations discussed in the previous section.

5.1 The Order Scan of QGM

As mentioned earlier, interesting orders are generated during the order scan, which takes place prior to the planning phase of optimization. Interesting orders arise from joins, ORDER BY, GROUP BY, or DISTINCT, and are hung off the QGM.

Each QGM box has an associated output order requirement, and each QGM quantifier has an associated input order requirement. In contrast to an interesting order, an *order requirement* forces a stream to have a specific order. Either the input or output order requirement can be empty. Output order requirements come from ORDER BY, while input order requirements currently come from GROUP BY. (Note that this does not preclude hash-based GROUP BY from being considered during the planning phase of optimization.) Each QGM box also has an associated list of interesting orders, which can double as sort-ahead orders.

Conceptually, the order scan has four stages. In the first stage, input and output order requirements are determined for each QGM box. Then, interesting orders for each DISTINCT is determined. Next, interesting orders for merge-joins and subqueries are determined. Finally, the QGM graph is traversed in a top-down manner.

In the top-down traversal, interesting orders are recursively pushed down along quantifier arcs. When an interesting order is pushed down to a quantifier Q, it gets homogenized to Q's columns and then covered with Q's input order requirement, if any. Similarly, before an interesting order can be pushed into a box B and added to B's list of interesting orders, it gets covered with B's output order requirement.

One subtlety in the order scan is that the algorithms for Cover Order and Homogenize Order require their inputs to be reduced. This in turn requires a set of applied predicates and FDs. Unfortunately, these are not known in the order scan since they are computed as properties during the planning phase of optimization.

This problem is resolved by proceeding optimistically. When an interesting order I is pushed down, the order scan simply assumes that all the predicates below a given box have been applied. Furthermore, if I cannot be fully homogenized to a quantifier, the largest prefix

of I that can be homogenized is used. This is done in the hope that some FD will make the suffix redundant. The planning phase can detect when these assumptions turn out to be false.

5.2 The Planning Phase of Optimization

During the planning phase of optimization, the DB2 optimizer walks the QGM bottom-up, box-by-box, and incrementally builds a QEP. For each box, alternative subplans are generated, and more costly subplans with comparable properties are pruned [Loh88]. The input and output interesting orders associated with each box are used to detect when a sort is required.

As a QEP is built, the interesting orders that hang off a QGM box are used for both pruning and to generate sort-ahead orders. During join enumeration, for example, the optimizer will try sorting the outer for each interesting order it finds. This allows a sort for, say, an ORDER BY to be pushed down an arbitrary number of levels in a join tree or view. If no sort is actually required at any level, this will be detected, of course. Note that this is only done for join methods where the order of the outer stream is propagated by the join.

When an interesting order is pushed down to the outer of a join, it has to be homogenized to the quantifier(s) that belong to the outer. This cannot be done during the order scan, since the order in which joins are enumerated is not known then. In the case of a merge-join, a cover with the merge-join order is also required.

Unfortunately, the process of pushing down sort-ahead orders increases the complexity of join enumeration [OL90]. This is because two join subtrees with the same tables but different orders are not compared and pruned against each other. It is possible to show that the complexity of join enumeration increases by a factor of $O(n^2)$ for n sort-ahead orders. In practice, this has not been problem, since typically $n \leq 3$.

5.2.1 Properties

For order optimization, the most important properties are the order property, the predicate property, the key property, and the FD property. Each of these is discussed in detail below. For any property x, the two primary ssues are how x propagates through operators and how two plans are compared on the basis of x.

How the different properties propagate will be discussed shortly. In terms of the way properties are compared, the DB2 optimizer treats everything uniformly. Let P_1 and P_2 be two plans being compared. Also, overload the symbol "$\leq$" for properties to mean less general or equivalent. Then P_2 prunes P_1 if $P_2.cost \leq P_1.cost$ and for every property x, $P_1.x \leq P_2.x$. In other words, P_1 can be pruned if it costs more than P_2 and has less general properties. Thus, for pruning, it suffices to define $\leq$ for each property.

The Order Property

The order property (if any) of a stream always originates from an ordered index scan or a sort. The way it propagates for most relational operators is straightforward except for projections and joins. If any column c_i of an order property $OP = (c_1, c_2, ..., c_n)$ is projected, then only the prefix $OP' = (c_1, c_2, ..., c_{i-1})$ is propagated.

For both nested-loops and merge-join [BE76], the order of the outer stream is propagated. In the special case when the outer has only one record, however, the inner order is propagated. There are also circumstances where the outer and inner orders can be concatenated, but that discussion is beyond the scope of this paper. For hash-join [DKO+84], neither the outer nor inner order is propagated.

The Test Order algorithm given in Section 4.2 is used to compare the order properties OP_1 and OP_2 of two plans during pruning. Let $int(OP_1)$ denote OP_1 cast as an interesting order. Then "$\leq$" can be defined for the order property as follows: $OP_1 \leq OP_2$ if OP_2 satisfies $int(OP_1)$.

The Predicate Property

The predicate property is simply the set of conjuncts which have been applied to a stream. Each operator propagates the predicate property by taking the predicate property of its input stream and unioning it with any conjuncts applied by the operator. For the predicate property, "$\leq$" is defined as follows: Let PP_1 and PP_2 be the predicate properties of two plans being compared. Then, $PP_1 \leq PP_2$ if $PP_1 \subseteq PP_2$.

The predicate property is used to determine both column equivalences and functional dependencies that arise from the application of equality predicates. In the DB2 optimizer, FDs that arise from predicates are not actually added to the FD property, however, since this information would be redundant and would only add to the complexity of maintaining the FD property (see below).

The Key Property

The key property of a stream is the set of unique keys for the stream. Each key K is represented as a set of columns $K = \{c_1, c_2, ..., c_n\}$. Keys are useful for a variety of reasons beyond their role in order optimization. One example is their use in DISTINCT elimination [PL94]. Consequently, in the DB2 optimizer, keys are maintained as a separate property.

Keys originate from base-table constraints or can be added via a GROUP BY or DISTINCT operation. If

any column c_i of a key $K = \{c_1, c_2, ..., c_n\}$ in a key property KP is projected by an operator, then K is removed from KP.

Whether a key propagates in a join requires analysis of the join predicates and the keys of the join's input streams. Consider the join of two streams S_1 and S_2 on join predicates JP. Let the key properties of S_1 and S_2 be denoted as KP_1 and KP_2 respectively. If a given row of S_1 can match at most one row of S_2 (i.e., the join is n-to-1), then KP_1 is propagated. This is true if any key $K = \{c_1, c_2, ..., c_n\}$ of KP_2 is *fully qualified* by predicates in JP of the form $S_1.col = S_2.c_i$ for all c_i. Similarly, if the join is 1-to-n, then KP_2 is also propagated. If neither KP_1 nor KP_2 can be propagated, then the key property of the join is formed by generating all concatenated key pairs $K_1 \cdot K_2$, where $K_1 \in KP_1$ and $K_2 \in KP_2$. For example, if $K_1 = \{a_1, a_2, ..., a_n\}$ and $K_2 = \{b_1, b_2, ..., b_m\}$ then $K_1 \cdot K_2 = \{a_1, a_2, ..., a_n, b_1, b_2, ..., b_m\}$.

An attempt is made to keep each key property as "succinct" as possible by removing keys that have become redundant because of projections and/or applied predicates. Each key is rewritten in a canonical form by substituting each column with its equivalence class head and removing redundant columns. If the DB2 optimizer detects that some key has become fully qualified by equality predicates during this process, then the entire key property is discarded and a *one-record condition* is flagged. This condition serves as the key property and indicates that at most one record is in the stream.

After simplifying each key in the property, redundant keys are removed from the key property using the definition of "$\leq$" that follows: Let key $K_1 = \{a_1, a_2, ..., a_n\}$ and let key $K_2 = \{b_1, b_2, ..., b_m\}$. Then $K_1 \leq K_2$ if $\{b_1, b_2, ..., b_m\} \subseteq \{a_1, a_2, ..., a_n\}$. If $K_1 \leq K_2$, then K_1 is implied by K_2. In that case, K_1 is redundant and can be removed.

The definition of "$\leq$" is also used to compare the key properties KP_1 and KP_2 of two plans during pruning. More specifically, $KP_1 \leq KP_2$ if for all $K_1 \in KP_1$ there exists some $K_2 \in KP_2$, where the relationship $K_1 \leq K_2$ holds. In other words, each $K_1 \in KP_1$ is implied by some $K_2 \in KP_2$.

The FD Property

In the DB2 optimizer, the FD property is simply a set of FDs, which can be empty. Each FD originates from a key. A key becomes an FD when it fails to propagate through a join. The columns of the key become the new FD's head, and the remaining columns in the key's stream become the new FD's tail. As an example, assume $K = \{c_1\}$ is a key in the join stream S with columns $\{c_1, c_2, ...c_n\}$. Further assume that the key property KP of S does not propagate in the join. Then, $\{c_1\} \rightarrow \{c_2, ..., c_n\}$ is added to the FD property of the join. The same is done for all keys in KP. Note that if S had a one-record condition, then the empty-headed FD $\{\} \rightarrow \{c_1, c_2, ..., c_n\}$ would be generated.

The effect of projection on the FD property is similar to the effect of projection on the key property. Let $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_m\}$. Then let F be a member of the FD property FP, where F is defined as $A \rightarrow B$. If any column a_i in A is projected, then F is removed from FP. In contrast, if any column b_i in B is projected, then F' replaces F, where F' is identical to F but with b_i removed from B.

Except for projection, FDs almost always propagate unchanged. In a join, the FDs of the outer and inner stream are combined and keys that do not propagate are used to generate new FDs, as described above. The resulting set of FDs can then be used to infer still more FDs [DD92]. This is not done in the DB2 optimizer because of its complexity, which is NP-complete in the general case [BB79].

Like the key property, an attempt is made to keep each FD property as "succinct" as possible by removing FDs that have become redundant because of projections and/or applied predicates. First, each FD is rewritten in a canonical form by substituting each column with its equivalence class head and removing redundant columns from both the head and tail. Then redundant FDs are removed from the FD property using the definition of "$\leq$" that follows: Let F_1 be defined as $A_1 \rightarrow B_1$ and let F_2 be defined as $A_2 \rightarrow B_2$. Then, $F_1 \leq F_2$ if $A_2 \subseteq A_1$ and $B_2 \supseteq B_1$. If $F_1 \leq F_2$, then F_1 is implied by F_2. In that case, F_1 is redundant and can be removed from the FD property.

The definition of "$\leq$" is also used to compare the FD properties FP_1 and FP_2 of two plans during pruning. More specifically, $FP_1 \leq FP_2$ if for all $F_1 \in FP_1$ there exists some $F_2 \in FP_2$, where the relationship $F_1 \leq F_2$ holds. In other words, each $F_1 \in FP_1$ is implied by some $F_2 \in FP_2$.

6 An Example

An example that illustrates how some of the techniques tie together is shown in Figure 6. In the example, the ORDER BY's interesting order $OB = (a.x)$ was pushed down and covered with the GROUP BY's interesting order $GB = (a.x, a.y, b.y)$. The resulting cover was then pushed down and itself covered with the merge-join's interesting order $MJ = (b.x)$. The key on $b.x$ gives rise to $\{b.x\} \rightarrow \{b.y\}$, which propagates through all the joins. This FD and the equivalence class generated by the predicate $a.x = b.x$ allowed the optimizer to detect that GB can be reduced to $GB = (a.x, a.y)$. As a result, the sort on $a.x, a.y$ simultaneously satisfies all interesting orders.

As shown, the optimizer determined that pushing

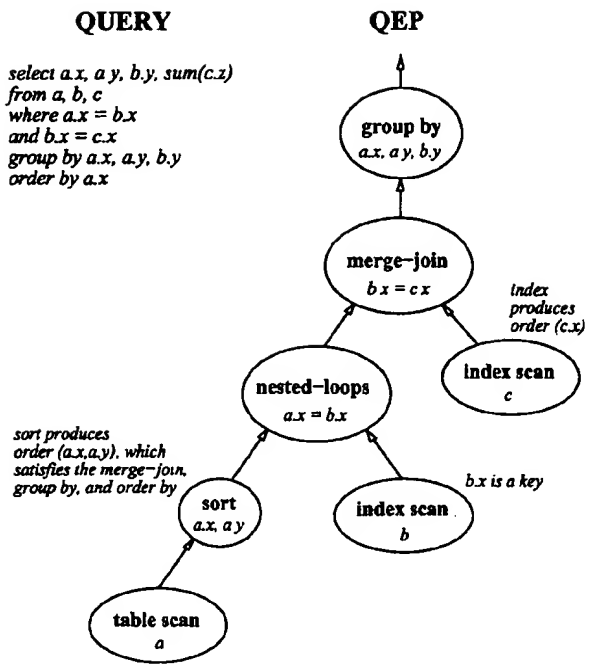


Figure 6: Query Example

down the sort before the first join results in the most efficient QEP. This is likely to be true if the size of table a is smaller than the result of either join. Because of the indexes on $b.x$ and $c.x$, the resulting QEP would probably beat one that used hash-based operators. Finally, note that the sort could be eliminated if there was an ordered index on $a.x, a.y$.

7 Advanced Issues

One of the issues that we have tacitly avoided in this paper is the fact that the order-based GROUP BY and DISTINCT operators do not dictate an exact interesting order. For example, consider a GROUP BY for $x, y, sum(distinct\ z)$. This can be satisfied by (x, y, z) or (y, x, z). Moreover, x, y, and z can be in ascending or descending order. In fact, a total of sixteen different orders can satisfy the order-based GROUP BY.

Rather than generate sixteen different interesting orders, one general interesting order is used in the real implementation. It includes information about which columns can be permuted and which columns can be in ascending or descending order. Using this information, the DB2 optimizer can correctly detect any order that satisfies the order-based GROUP BY. Accounting for these "degrees of freedom" adds a non-trivial amount of complexity to all operations on orders. It probably doubled the amount of code. In general, though, the same underlying logic that has been described in this paper still prevails.

8 Performance Results

Clearly, the techniques described in this paper for order optimization can only improve the quality of execution plans produced by an optimizer. In cases where an execution plan's performance would degrade, which can happen with sort-ahead, an optimizer would simply pick a better alternative using its cost estimates. Therefore, the only question is whether the improvement in performance offered by our techniques is worth the implementation effort. More specifically, are there a lot of "real world" queries where the improvement in performance is significant?

IBM maintains a number of internal benchmarks that have been inspired by real DB2 customers over the years. On those benchmarks and at customer sites, we have observed substantial improvement in the performance of many queries because of the techniques described in this paper. The biggest improvements are typically seen in decision-support environments with lots of indexes. Often, applications in these environments cannot fully anticipate the predicates that will be specified by end-users at runtime. Nor can they anticipate schema changes, such as the addition of a new index or key. As a result, queries in these environments frequently include a lot of redundancy – grouping on key columns, sorting on columns that are bound to constants through predicates, and so on. Order optimization is able to eliminate this kind of redundancy, which in turn usually leads to a better exection plan.

8.1 TPC-D Results

Unfortunately, the benchmarks described above are unknown outside of IBM. Therefore, we turn to the TPC-D benchmark [1] to illustrate how much our techniques for order optimization can improve performance. A description of the TPC-D benchmark and its schema is omitted. For details, readers are directed to [Eng95].

TPC bylaws prohibit us from disclosing a full set of unaudited TPC-D results. Moreover, IBM was reluctant to let certain results be published when this paper was written. Consequently, the focus here will be on just Query 3 of the TPC-D benchmark. Query 3 was chosen because it is (relatively) simple and benefits from several of the techniques that have been described in this paper. Query 3 retrieves the shipping priority and potential revenue of the orders having the largest revenue among those that had not been shipped as of a given date. It is defined as follows:

[1] TPC-D is a trademark of the Transaction Processing Concil.

```
select l_orderkey,
   sum(l_extendedprice * (1 - l_discount)) as rev,
   o_orderdate, o_shippriority
from  customer, order, lineitem
where o_orderkey = l_orderkey
and   c_custkey = o_orderkey
and   c_mktsegment = 'building'
and   o_orderdate < date('1995-03-15')
and   l_shipdate > date('1995-03-15')
group by l_orderkey, o_orderdate, o_shippriority
order by rev desc, o_orderdate
```

To gather performance results, we built a modified version of DB2 with order optimization disabled. Then we ran queries on both the production and disabled version of DB2. Results were obtained on a 1GB TPC-D database using a single IBM RS/6000 Model 59H (66 Mhz) server with 512MB of memory and running AIX 4.1. A real benchmark configuration was used, with data striped over 15 disks and 4 I/O controllers. Using a combination of big-block I/O, prefetching, and I/O parallelism, this configuration was able to drive the CPU at 100% utilization.

The results for Query 3 are shown in Table 1. The numbers in the table correspond to the elapsed time to run Query 3, averaged over five runs. As shown, the elapsed time for the version of DB2 with order optimization disabled was significantly slower than the production version of DB2 (by a ratio of 2.04).

Production DB2	Disabled DB2	Ratio
192 sec.	393 sec.	2.04

Table 1: Elapsed Time for Query 3

The execution plan chosen by the production version of DB2 is shown in Figure 7. Using a combination of Reduce Order, Cover Order, and Homogenize Order, the DB2 optimizer was able to determine that it was beneficial to push the sort for the GROUP BY below the nested-loop join. This sort not only provided the required order for the GROUP BY, but it also caused the index probes in the nested-loop join to become clustered. We refer to these as *ordered* nested-loop joins. Here, an ordered nested-loop join is especially important because it allows prefetching and parallel I/O to be used on the *lineitem* table, which is the largest of all the TPC-D tables.

In Figure 7, note that the sort on *o_orderkey* satisfied the GROUP BY because of the equivalence class generated by the predicate $o\_orderkey = l\_orderkey$ and because of the FD $\{o\_orderkey\} \rightarrow \{o\_orderdate, o\_shippriority\}$. In SQL queries, there is often no choice but to include functionally dependent

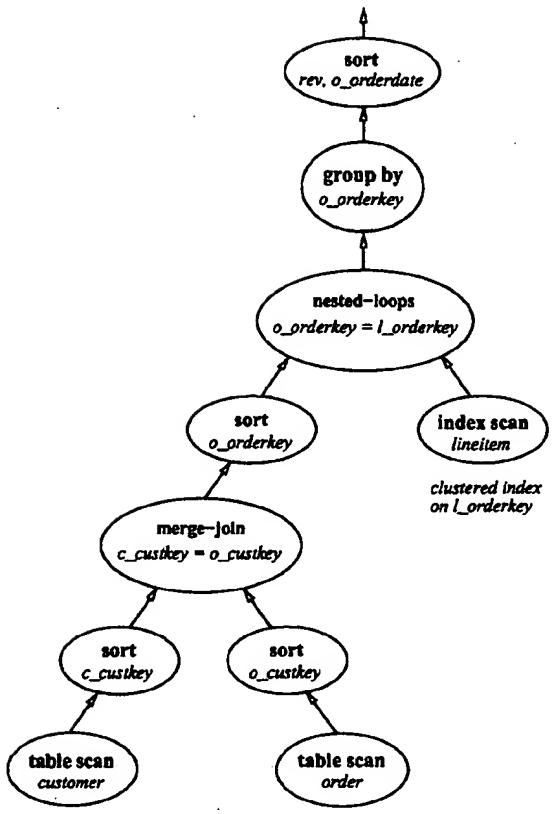


Figure 7: Query 3 in Production Version of DB2

(i.e., redundant) columns like these in a GROUP BY, since that is the only way to have them appear as output.

For comparison, the execution plan chosen by the version of DB2 with order optimization disabled is shown in Figure 8. In this case, the DB2 optimizer was unable to detect that the sort on *o_orderkey* satisfies the GROUP BY. Moreover, without an awareness of equivalence classes, the optimizer was unable to determine that the same sort could be used to generate an ordered nested-loop join for the *lineitem* table. Consequently, a more costly merge-join was used.

9 Conclusion

This paper described the novel techniques that are used for order optimization in the query optimizer of IBM's DB2. These general techniques, which can be used by any query optimizer, make it possible to detect when sorting can be avoided because of predicates, keys, indexes, or functional dependencies; the minimal number of sorting columns when a sort is unavoidable; whether a sort can be pushed down into a view or join tree to make it cheaper; and whether two or more sorts can be

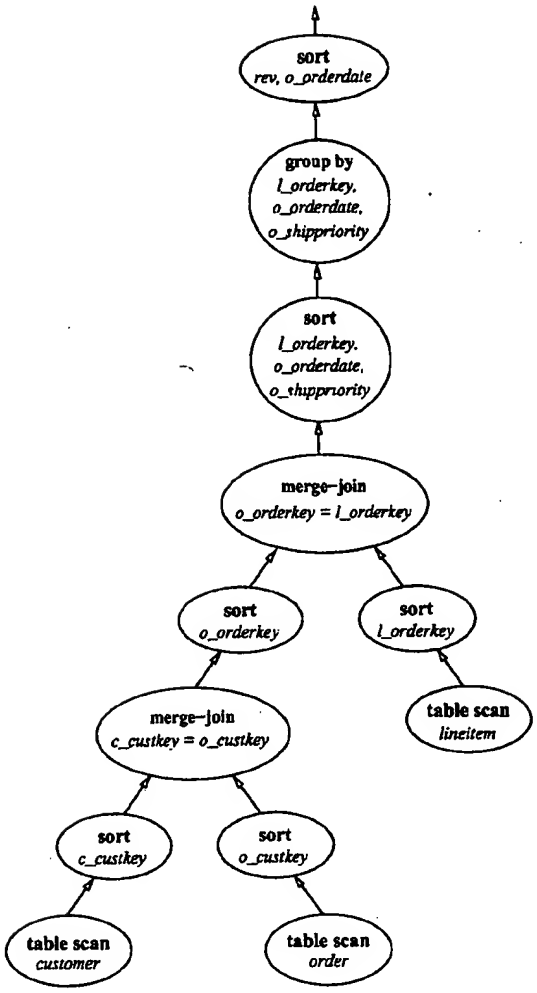


Figure 8: Query 3 with Order Optimization Disabled

combined and satisfied by a single sort. For complex queries in a data warehouse environment, these techniques can mean the difference between an execution plan that finishes in a few minutes verses one that takes hours to run.

This paper's main contribution was a set of fundamental operations for use in order optimization. Algorithms were provided for testing whether an interesting order is satisfied, for combining two interesting orders, and for pushing down an interesting order in a query graph. All of these hinge on a core operation called *Reduce Order*, which uses functional dependencies and predicates to reduce interesting orders to a simple canonical form.

This paper also described the overall architecture of the DB2 optimizer for order optimization. In particular, the paper described how order, predicates, keys, and functional dependencies can be maintained as access plan properties. The importance of maintaining functional dependencies as a property goes beyond just order optimization. Functional dependencies can be used for other optimizations as well [DD92].

Finally, results for Query 3 of the TPC-D benchmark were provided to illustrate how much the techniques described in this paper can improve performance. On a 1GB TPC-D database, a version of DB2 with order optimization disabled ran Query 3 roughly 2x slower than the production version of DB2 with order optimization enabled.

Acknowledgements

The authors would like to thank Bobbie Cochrane, Guy Lohman, and Jeff Naughton for reading earlier drafts of this paper. Thanks also go to Bernie Schiefer for generating TPC-D benchmark results.

References

[Ant93] G. Antosheknov. Query processing in dec rdb: Major issues and future challenges. In *IEEE Bulletin on the Technical Comittee on Data Engineering*, December 1993.

[BB79] C. Beeri and P. Bernstein. Computational problems related to the design of normal form relational schemas. In *ACM Transactions on Database Systems*, March 1979.

[BD83] D. Bitton and D. DeWitt. Duplicate record elimination in large data files. In *ACM Transactions on Database Systems*, June 1983.

[BE76] M. Blasgen and K. Eswaran. On the evaluation of queries in a relational data base system. Technical Report 1745, IBM Santa Teresa Lab, April 1976.

[CS93] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 19th International Conference on Very Large Data Bases*, August 1993.

[DD92] H. Darwen and C. Date. The role of functional dependencies in query decomposition. In *Relational Database Writings 1989-1991*. Addison Wesley, 1992.

[DKO+84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, June 1984.

[Eng95] S. Englert. Tpc benchmark d. In *Transaction Processing Performance Council*, 777 N. First St, Suite 600, San Jose CA 95112-6311, October 1995.

[GD87] G. Graefe and D. DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, June 1987.

[Gra93] G. Graefe. Query evaluation techniques for large databases. In *ACM Computing Surveys*, June 1993.

[Hel94] J. Hellerstein. Pratical predicate placement. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, June 1994.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, June 1989.

[JV84] M. Jarke and Y. Vassiliou. Query optimization in database systems. In *ACM Computing Surveys*, June 1984.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, June 1988.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, August 1990.

[PHH92] H. Pirahesh, J. Hellstein, and W. Hasan. Extensible rule based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, June 1992.

[PL94] G. Paulley and P. Larson. Exploiting uniqueness in query optimization. In *International Conference on Data Engineering*, February 1994.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, June 1979.

[YL93] P. Yan and P. Larson. Performing group-by before join. In *International Conference on Data Engineering*, February 1993.